ESC-TR-2006-071

Project Report
SPR-9

Computational Workloads for Commonly Used Signal Processing Kernels

M. Arakawa

28 May 2003
Reissued: 30 November 2006

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

Prepared for the Office of the Secretary of Defense under Air Force Contract FA8721-05-C-0002.

This report is based on studies performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored by the Office of the Secretary of Defense under Air Force Contract FA8721-05-C-0002.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Gary Tutungian
Administrative Contracting Officer
Plans and Programs Directorate
Contracted Support Management

Massachusetts Institute of Technology
Lincoln Laboratory

Computational Workloads for Commonly Used Signal Processing Kernels

M. Arakawa
Group 102

Project Report SPR-9

28 May 2003

Reissued: 30 November 2006

Lexington Massachusetts

ABSTRACT

In the course of designing or evaluating signal processing algorithms, we often must determine the computational workload needed to implement the algorithms on a digital computer. The floating-point operation (flop) counts for real versions of the most common signal processing kernels are well documented. However, the flop counts for kernels operating on complex inputs are not as readily found. This report collects the flop count expressions for both real and complex kernels and also presents brief outlines of the derivations for the flop count expressions. These flop count expressions are summarized below.

Signal Processing Kernel	**Computational Complexity**	
	Real Input	**Complex Input**
matrix-matrix multiplication	$2mnp$	$8mnp$
fast Fourier transform	$\frac{5}{2}n\log_2 n$	$5n\log_2 n$
Householder QR decomposition	$2n^2\left(m-\frac{n}{3}\right)$	$8n^2\left(m-\frac{n}{3}\right)$
forward or back substitution	n^2	$4n^2$
eigenvalue decomposition: eigenvalues only	$\frac{4}{3}n^3$	$\frac{16}{3}n^3$
eigenvalue decomposition: eigenvalues and eigenvectors	$9n^3$	$23n^3$
singular value decomposition: singular values only	$4mn^2-\frac{4}{3}n^3$	$16mn^2-\frac{16}{3}n^3$
singular value decomposition: singular values and left singular vectors	$4m^2n+12mn^2$	$16m^2n+24mn^2$
singular value decomposition: singular values and right singular vectors	$4mn^2+12n^3$	$16mn^2+24n^3$
singular value decomposition: singular values, left and right singular vectors	$4m^2n+12mn^2+13n^3$	$16m^2n+24mn^2+29n^3$

In the table above, the parameters in the computational complexity expressions are:

- the dimensions of the two multiplicands - $m \times n$ and $n \times p$ - for the matrix-matrix multiplication
- the length of the vector n for the fast Fourier transform
- the size of the triangular system n for forward and back substitutions
- the dimensions of the input matrix $m \times n$ for the Householder QR decomposition, eigenvalue decomposition, and singular value decomposition.

ACKNOWLEDGMENTS

In pulling all of this material together, I learned a fair bit about linear algebra. I would like to thank Charlie Rader for sharing his knowledge on and insight into this topic and for his review of this document. I also benefitted from many conversations with James Lebak on this subject. I would also like to thank Ed Baranoski and Bob Bond for giving me the time to write this report.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

In the course of designing or evaluating signal processing algorithms, we must often determine the computational workload needed to implement the algorithms on a digital computer. The floating-point operation (flop) counts for real versions of the most common signal processing kernels are well documented. However, the flop counts for kernels operating on complex inputs are not as readily found. This report collects the flop count expressions for both real and complex kernels and also presents brief outlines of the derivations for the flop count expressions. Specifically, the following computational kernels will be treated:

- matrix-matrix multiplication
- fast Fourier transform (FFT)
- Householder QR factorization
- forward and back substitutions
- eigenvalue decomposition
- singular value decomposition

2. MATRIX MULTIPLICATION

2.1 REAL MATRIX MULTIPLICATION

The elements of the product matrix $C \in \mathcal{R}^{m \times n}$ of real matrices $A \in \mathcal{R}^{m \times p}$ and $B \in \mathcal{R}^{p \times n}$ are given by ([4]):

$$C = AB \Rightarrow c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \tag{1}$$

Therefore, for each of the mn elements in C, we perform p multiplications and p additions, giving us a flop count of:

$$\text{flop count} = 2mnp \tag{2}$$

2.2 COMPLEX MATRIX MULTIPLICATION

The elements of the product matrix $C \in \mathcal{C}^{m \times n}$ of complex matrices $A \in \mathcal{C}^{m \times p}$ and $B \in \mathcal{C}^{p \times n}$ are given by ([4]):

$$C = AB \Rightarrow c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \tag{3}$$

For each of the mn elements in C, we perform p multiplications and p additions. However, these multiplications and additions are between complex numbers.

The complex sum of two complex scalars requires two flop: one flop to add the real components, and another flop to add the imaginary components.

The complex product z of two complex scalars $x = a + jb$ and $y = c + jd$, where a, b, c, and d are all real scalars and $j = \sqrt{-1}$ in this context, is:

$$\begin{aligned} z &= xy \\ &= (a + jb) \times (c + jd) \\ &= (ac - bd) + j \times (ad + bc) \end{aligned} \tag{4}$$

and requires six flop:

- four multiplications: ac, bd, ad, and bc
- two additions: $(ac - bd)$ and $(ad + bc)$

Therefore, for each of the mn elements in C, we perform $8p$ flop, giving us a flop count of:

$$\text{flop count} = 8mnp \tag{5}$$

2.3 ALTERNATIVE COMPLEX MATRIX MULTIPLICATION

We can compute the product matrix $C \in \mathcal{C}^{m \times n}$ of complex matrices $A \in \mathcal{C}^{m \times p}$ and $B \in \mathcal{C}^{p \times n}$ in an alternative fashion that reduces the workload by approximately 25% ([10]).

First, we separate the multiplicands A and B into their real and imaginary components. Let $A_r \in \mathcal{R}^{m \times p}$ be the real part of A, $A_i \in \mathcal{R}^{m \times p}$ be the imaginary part of A, $B_r \in \mathcal{R}^{p \times n}$ be the real part of B, and $B_i \in \mathcal{R}^{p \times n}$ be the imaginary part of B:

$$A = A_r + jA_i \tag{6}$$

$$B = B_r + jB_i \tag{7}$$

Then, the product matrix C is:

$$\begin{aligned} C &= AB \\ &= (A_r + jA_i) \times (B_r + jB_i) \\ &= (A_rB_r - A_iB_i) + j(A_rB_i + A_iB_r) \end{aligned} \tag{8}$$

Now consider the real matrices $G \in \mathcal{R}^{m \times p}$ and $H \in \mathcal{R}^{p \times n}$, which we define thusly:

$$G = A_r + A_i \tag{9}$$

$$H = B_r - B_i \tag{10}$$

Then

$$GH = A_rB_r - A_rB_i + A_iB_r - A_iB_i \tag{11}$$

$$GH + A_rB_i - A_iB_r = A_rB_r - A_iB_i \tag{12}$$

$$\begin{aligned} GH + A_rB_i - A_iB_r + j(A_rB_i + A_iB_r) &= A_rB_r - A_iB_i + j(A_rB_i + A_iB_r) \\ &= C \end{aligned} \tag{13}$$

Computing $G = A_r + A_i$ requires mp flop. Computing $H = B_r - B_i$ requires pn flop. Computing the product GH requires $2mnp$ flop. Computing the product A_rB_i also requires $2mnp$. Computing the product A_iB_r requires another $2mnp$ flop. Computing $GH + A_rB_i - A_iB_r$ for the real portion of C requires $2mn$ flop. Computing $A_rB_i + A_iB_r$ for the imaginary portion of C requires mn flop. The total flop count for computing $C = AB$ using this alternative method is

$$\text{flop count} = 6mnp + 3mn + (m+n)p \tag{14}$$

which is approximately three-quarters of the $8mnp$ flop needed to compute the product using the straightforward method.

3. FAST FOURIER TRANSFORM

3.1 COMPLEX FFT

The discrete Fourier transform (DFT) X of a finite-length sequence x of length N is given by ([7]):

$$X[k] = \sum_{n=0}^{N-1} x[n]{W_N}^{kn}, \; k = 0, 1, \ldots, N-1 \tag{15}$$

where

$$W_N = e^{-j(2\pi/N)} \tag{16}$$

If we were to compute the DFT by explicitly evaluating the sums, the flop count would be $O(N^2)$. We can dramatically reduce the workload by decomposing the original DFT into successively smaller DFT computations in decimation-in-time algorithms ([7]). Let us consider the case where N is a power of two: $N = 2^{\upsilon}$, where υ is an integer.

Since N is an even integer, we can separate $x[n]$ into two sequences of length $N/2$: the first sequence consists of the even-numbered points in $x[n]$, while the second sequence consists of the odd-numbered points in $x[n]$. We can now rewrite Equation 15 as:

$$X[k] = \sum_{n \text{ even}} x[n]{W_N}^{kn} + \sum_{n \text{ odd}} x[n]{W_N}^{kn} \tag{17}$$

Let us substitute $n = 2r$ for even n and $n = 2r+1$ for odd n:

$$\begin{aligned} X[k] &= \sum_{r=0}^{(N/2)-1} x[2r]{W_N}^{2rk} + \sum_{r=0}^{(N/2)-1} x[2r+1]{W_N}^{(2r+1)k} \\ &= \sum_{r=0}^{(N/2)-1} x[2r]({W_N}^2)^{rk} + {W_N}^k \sum_{r=0}^{(N/2)-1} x[2r+1]({W_N}^2)^{rk} \end{aligned} \tag{18}$$

We also note that ${W_N}^2 = W_{N/2}$:

$$ {W_N}^2 = e^{-2j(2\pi/N)} = e^{-j2\pi/(N/2)} = W_{N/2} \tag{19}$$

We can therefore rewrite Equation 18:

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r]W_{N/2}^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1]W_{N/2}^{rk} \tag{20}$$

$$= G[k] + W_N^k H[k]$$

Both $G[k]$ and $H[k]$ are $(N/2)$-point DFTs, where $G[k]$ is the DFT of the even-numbered points of $x[n]$ and $H[k]$ is the DFT of the odd-numbered points of $x[n]$. If the length of the original sequence N is a power of 2, we can use this technique to continue to further decompose the DFTs until we have only DFTs of length 2 (see Figure 1 for the case of $N = 8$). We would have a total of $\nu = \log_2 N$ stages.

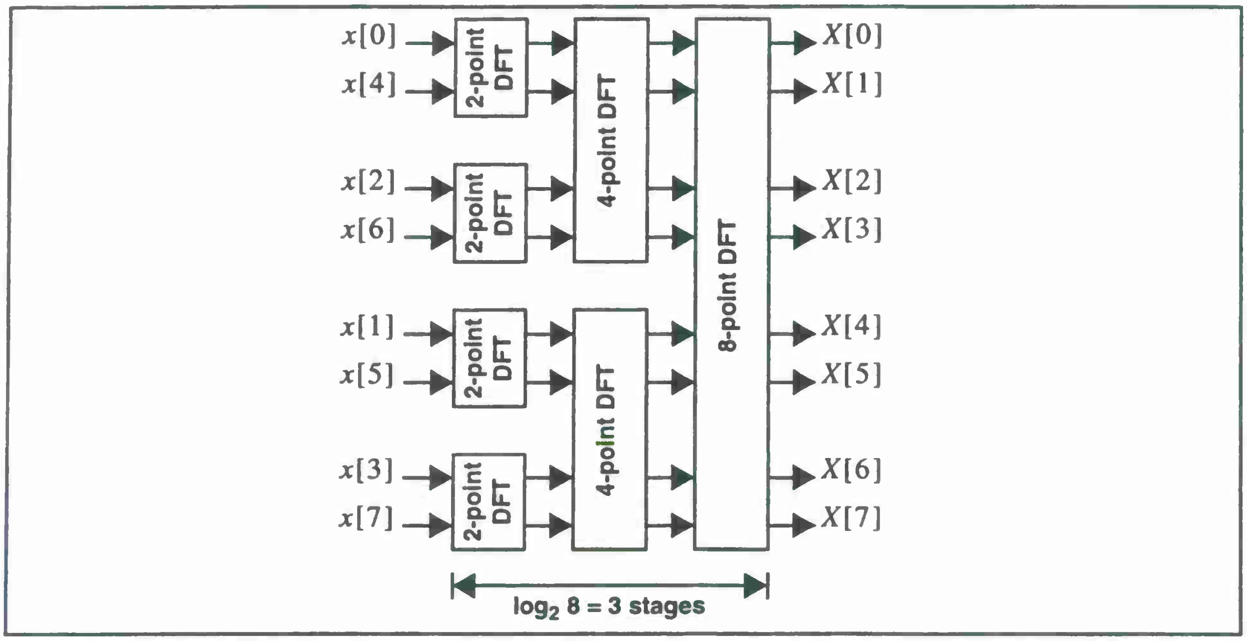

Figure 1: Decimation-in-time decomposition of an 8-point DFT into 2-point DFTs

If we expand Figure 1 with the final expression in Equation 20, we arrive at the flow graph shown below in Figure 2 (for the case $N = 8$):

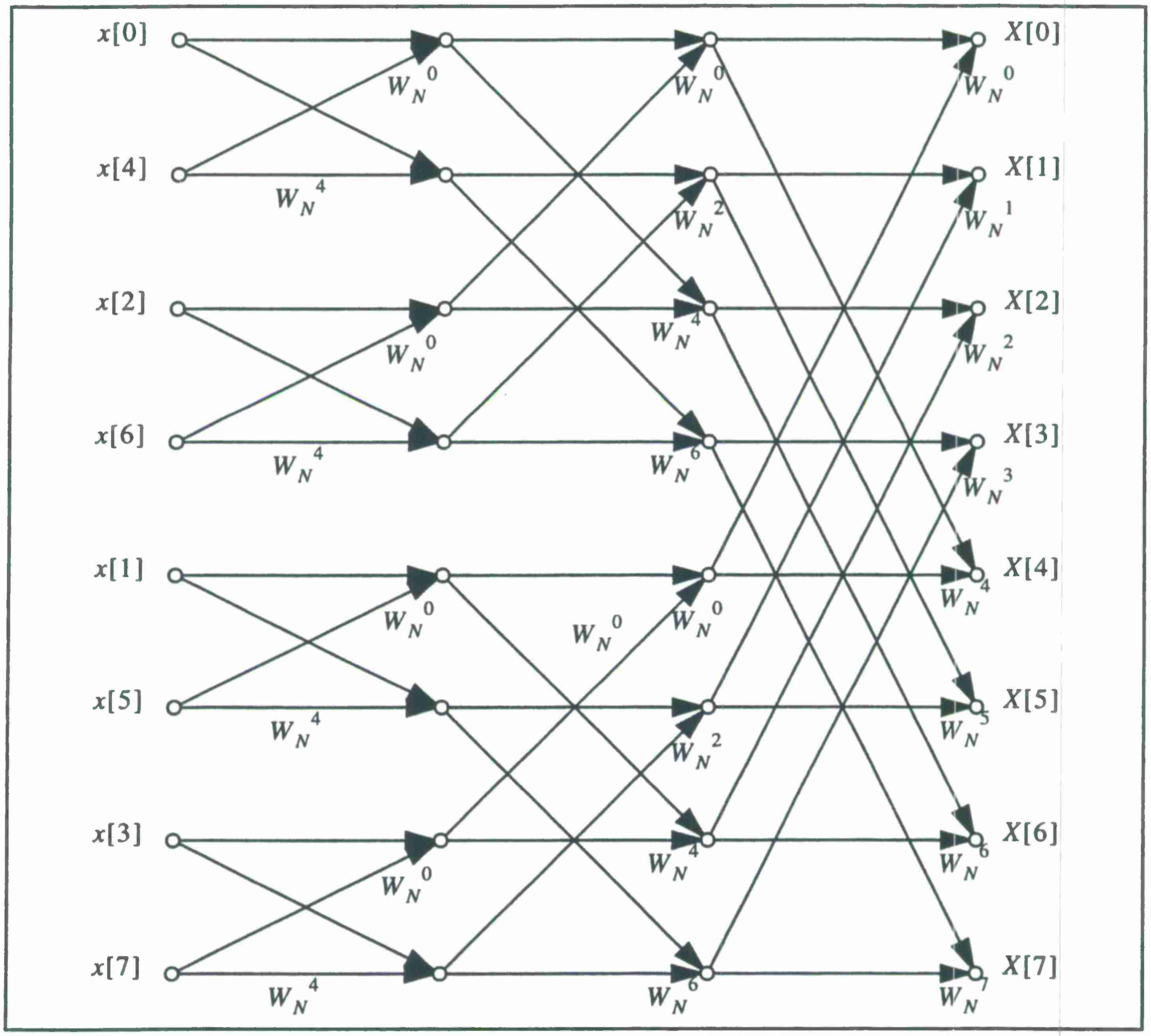

Figure 2: Flow graph of the complete decimation-in-time decomposition of an 8-point DFT ([7])

In Figure 2, we used a notation where branches (arrows) entering a node (circle) denotes the addition of the quantities from which the branches originated, and a coefficient next to the head of a branch denotes a scaling of the quantity by the coefficient. If no coefficient is indicated, the scaling factor is assumed to be unity by default. See Figure 3 for an illustration of this notation.

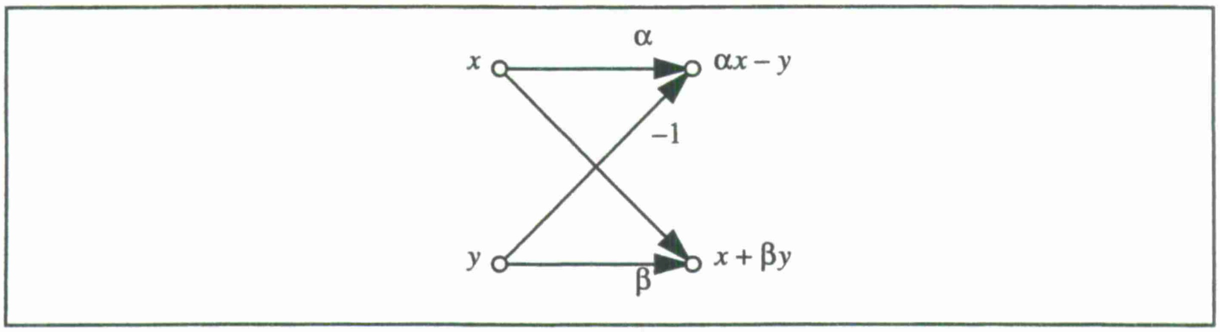

Figure 3: Flow graph notation

The fundamental operation at each pair of nodes, called a butterfly, is shown below in Figure 4.

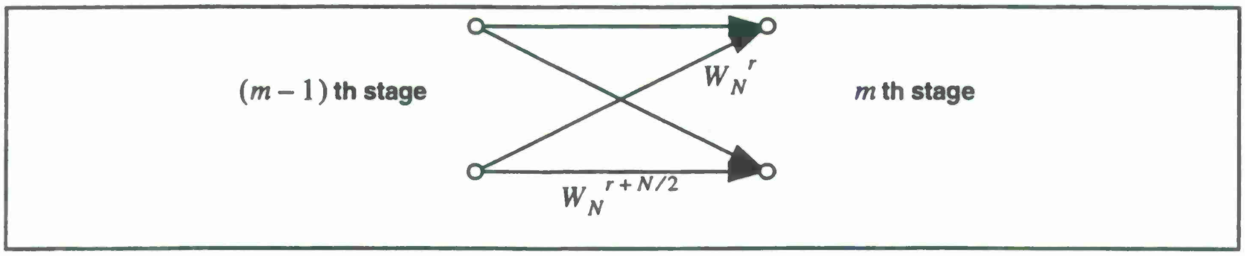

Figure 4: Flow graph of basic butterfly computation ([7])

At each non-input node, we perform a complex multiplication and a complex addition, which together require 8 flop. There are $\log_2 N$ stages of nodes, giving us a total flop count of $8N\log_2 N$. We can reduce this flop count further by noting that ([7])

$$W_N^{\,N/2} = e^{-j(2\pi/N)N/2} = e^{-j\pi} = -1 \tag{21}$$

and that therefore

$$W_N^{\,r+N/2} = W_N^{\,N/2} W_N^{\,r} = -W_N^{\,r} \tag{22}$$

The butterfly computation shown in Figure 4 can be simplified, as shown in Figure 5.

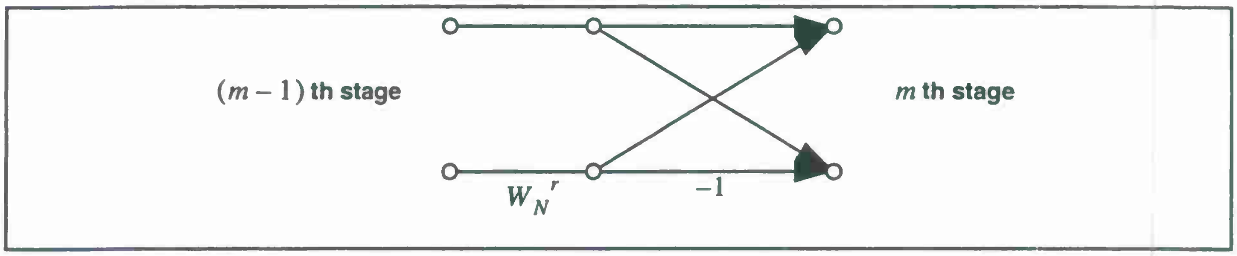

Figure 5: Simplified butterfly computation requiring only one complex multiplication ([7])

Using this identity, we will need to multiply by W_N^k half as many times, resulting in the flow graph shown below in Figure 6 (for the case $N = 8$).

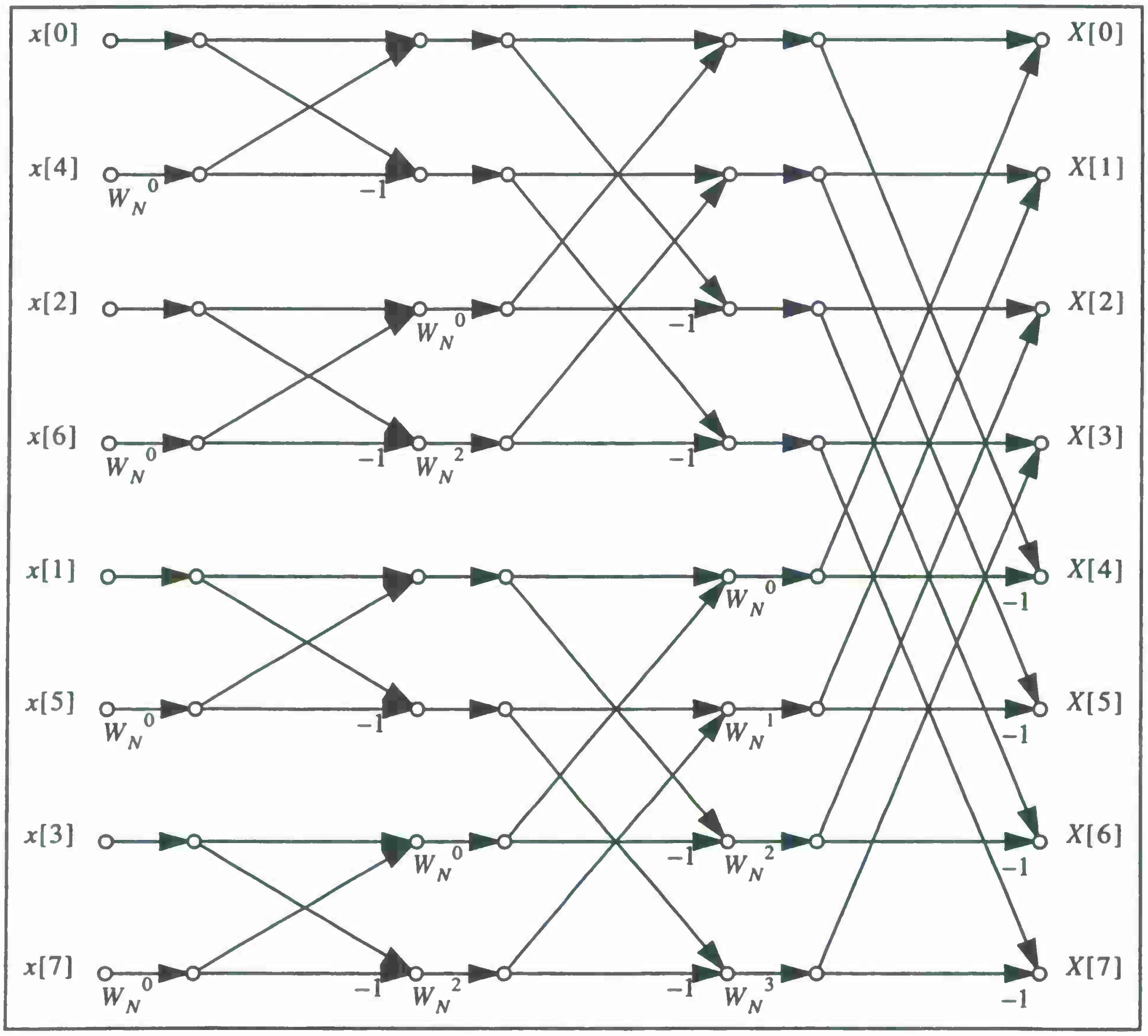

Figure 6: An 8-point DFT using the butterfly computation in Figure 5 ([7])

For each pair of nodes, we perform one multiply, one add, and one subtract, for a total of 10 flop per pair of nodes or an average of 5 flop per node:

$$\text{flop count} \cong 5N\log_2 N \tag{23}$$

Because W_N depends only on N, it can be pre-computed, and its evaluation does not contribute to the workload.

This technique is not limited to radix 2 FFTs; it may be applied to other radices.

3.2 REAL FFT

First, we note that the DFT of a real vector is conjugate even: if $x \in \mathcal{R}^N$, the DFT $X \in C^N$ of x has the properties that

$$Re\{X[k]\} = Re\{X[N-k]\} \quad (24)$$

$$Im\{X[k]\} = -Im\{X[N-k]\} \quad (25)$$

If we take advantage of the resulting additional structure, we can reduce the workload necessary to evaluate the FFT of a real vector.

The computation of the FFT of a real vector uses two algorithms. The first algorithm computes the DFT of two real vectors of equal length through a single FFT of a complex vector of the same length ([8]). This algorithm will be used to perform FFTs on the two halves of the input vector.

Let f and g be two real vectors of length N, where f consists of samples $f[0]$ through $f[N-1]$ and g consists of samples $g[0]$ through $g[N-1]$. Let F be the DFT of f and G be the DFT of g.

The fact that f is real means that $Re\{F\}$ (the real part of F) is an even function and $Im\{F\}$ (the imaginary part of F) is an odd function. An even function $Re\{F\}$ is defined as follows:

$$Re\{F[k]\} = Re\{F[N-k]\} \text{ for } k \neq 0 \quad (26)$$

An odd function $Im\{F\}$ is defined as follows:

$$Im\{F[0]\} = 0 \quad (27)$$

$$Im\{F[k]\} = -Im\{F[N-k]\} \text{ for } k \neq 0 \quad (28)$$

If g is real, then jg is imaginary, $Re\{jG\}$ is an odd function, and $Im\{jG\}$ is an even function:

$$Re\{jG[0]\} = 0 \quad (29)$$

$$Re\{jG[k]\} = -Re\{jG[N-k]\} \text{ for } k \neq 0 \quad (30)$$

$$Im\{jG[k]\} = Im\{jG[N-k]\} \text{ for } k \neq 0 \quad (31)$$

Let us define

$$h \equiv f + jg \quad (32)$$

Then H, the DFT of h, is

$$H = F + jG \quad (33)$$

Let us first consider the real part of H:

$$Re\{H\} = Re\{F\} + Re\{jG\} \tag{34}$$

where $Re\{F\}$ is even and $Re\{jG\}$ is odd. Given these conditions, we can determine that

$$Re\{jG[0]\} = 0 \rightarrow Re\{F[0]\} = Re\{H[0]\} \tag{35}$$

$$Re\{F[1:N-1]\} = \frac{Re\{H[1:N-1]\} + Re\{H[N-1:-1:1]\}}{2} \tag{36}$$

$$Re\{jG[1:N-1]\} = \frac{Re\{H[1:N-1]\} - Re\{H[N-1:-1:1]\}}{2} \tag{37}$$

where $H[N-1:-1:1]$ are elements 1 through $N-1$ of H in reverse order.

Next, let us consider the imaginary part of H:

$$Im\{H\} = Im\{F\} + Im\{jG\} \tag{38}$$

where $Im\{F\}$ is odd and $Im\{jG\}$ is even. Given these conditions, we can determine that

$$Im\{F[0]\} = 0 \rightarrow Im\{jG[0]\} = Im\{H[0]\} \tag{39}$$

$$Im\{F[1:N-1]\} = \frac{Im\{H[1:N-1]\} - Im\{H[N-1:-1:1]\}}{2} \tag{40}$$

$$Im\{jG[1:N-1]\} = \frac{Im\{H[1:N-1]\} + Im\{H[N-1:-1:1]\}}{2} \tag{41}$$

Of course, once we have jG, we can compute G:

$$G = -jG \times j \tag{42}$$

or

$$G = Im\{jG\} - j \times Re\{jG\} \tag{43}$$

A MATLAB implementation of this algorithm is shown below in Figure 7.

```
% f and g are the real input vectors

h = f + i*g;
H = fft(h);

F(1) = real(H(1));
F(2:N) = 0.5 * (real(H(2:N)) + real(H(N:-1:2)));
jG(2:N) = 0.5 * (real(H(2:N)) - real(H(N:-1:2)));

jG(1) = i * imag(H(1));
F(2:N) = F(2:N) + 0.5i * (imag(H(2:N)) - imag(H(N:-1:2)));
jG(2:N) = jG(2:N) + 0.5i * (imag(H(2:N)) + imag(H(N:-1:2)));

G = -jG * i;
```

Figure 7: MATLAB code for computing the DFT of two real vectors using one complex FFT

The workload for this algorithm is dominated by the FFT of the complex vector h, and is therefore approximately $5N\log_2 N$ flop.

The second algorithm needed to compute a real FFT applies a conjugate-even butterfly to an input vector ([11]). This algorithm is used to combine the two conjugate-even half-length vectors produced by the first algorithm to produce a single conjugate-even full-length vector. Specifically, if $x \in \mathcal{R}^L$, where $L = 2^q$ and $q \geq 1$, the algorithm in Figure 8 overwrites x with $B_L^{(ce)}x$, where $B_L^{(ce)}$ is the conjugate-even butterfly matrix for an L-point FFT.

```
if q ~= 1
  L_star = L / 2;
  p = L / 4;

  tau = x(1);
  x(1) = tau + x(L_star + 1);
  x(L_star + 1) = tau - x(L_star + 1);
  x(3 * p + 1) = -x(3 * p + 1);

  for count = 2:p
    c = cos(2 * pi * (count - 1) / L);
    s = sin(-2 * pi * (count - 1) / L);
    u(count) = c * x(L_star + count) - s * x(L_star + p + count);
    v(count) = s * x(L_star + count) + c * x(L_star + p + count);
  end  % for count

  y(2:p) = x(2:p);
  x(2:p) = x(p + 2:L_star);

  for count = 2:p
    x(count) = y(count) + u (count);
    x(p + count) = y(p - count) - u(p - count);
    x(L_star + count) = z(count) + v(count);
    x(L_star + p + count) = -z(p - count) + v(p - count)
  end  % for count

else
  tau = x(1);
  x(1) = tau + x(2);
  x(2) = tau - x(2);
end  % if q ~= 1
```

Figure 8: MATLAB code to apply a conjugate-even butterfly ([11])

This algorithm requires approximately $5L/2$ flop.

To compute the FFT X of a single real vector $x \in \mathcal{R}^n$ ([11]), the input vector is first divided into two half-length vectors: let $x_{odd} = x[1:2:n]$ and $x_{even} = x[2:2:n]$. Next, we use the algorithm in Figure 7 to compute $v_{odd}^{(ce)}$ and $v_{even}^{(ce)}$, which are the DFTs of x_{odd} and x_{even}, respectively. Finally, we use the algorithm in Figure 8 to compute $X = B_n^{(ce)} \begin{bmatrix} v_{odd}^{(ce)} \\ v_{even}^{(ce)} \end{bmatrix}$.

Computing the DFTs of the half-length vectors requires

$$\text{flop count} \cong 5\left(\frac{n}{2}\right)\log_2\left(\frac{n}{2}\right) \tag{44}$$

$$= \frac{5}{2}n[(\log_2 n) - 1]$$

$$= \left(\frac{5}{2}n\log_2 n\right) - \frac{5}{2}n$$

Applying the conjugate-even butterfly matrix requires an additional $\frac{5}{2}n$ flop, bringing the total flop count for an FFT of a real vector to

$$\text{flop count} \cong \frac{5}{2}n\log_2 n \tag{45}$$

4. HOUSEHOLDER QR DECOMPOSITION

4.1 REAL HOUSEHOLDER QR DECOMPOSITION

The QR decomposition factors a real input matrix $A \in \mathcal{R}^{m \times n}$ into an orthogonal matrix $Q \in \mathcal{R}^{m \times m}$ and an upper triangular matrix $R \in \mathcal{R}^{m \times n}$ ([4]):

$$A = QR \tag{46}$$

$$Q^T Q = I, QQ^T = I \tag{47}$$

The Householder QR factorization algorithm entails the application of a series of orthogonal transformations P_i to the input matrix A so that the portion of the matrix below the diagonal will become zero:

$$P_n \ldots P_1 A = R \tag{48}$$

where each matrix P_i has the form

$$P = I - \beta v v^T \tag{49}$$

To avoid explicitly computing the Householder reflection $I - \beta v v^T$, we use the following implementation ([4]):

$$\text{define } w = \beta A^T v \tag{50}$$

$$\begin{aligned}(I - \beta v v^T)A &= A - \beta v v^T A \\ &= A - v(\beta v^T A) \\ &= A - v w^T\end{aligned} \tag{51}$$

A MATLAB implementation of the real Householder QR decomposition, which overwrites the input matrix with the upper triangular factor (the Householder QR decomposition does not return the orthogonal factor), is shown below in Figure 9.

```
[num_rows, num_cols] = size(A);
for col = 1:num_cols
  % Compute the Householder vector v
  [v, beta] = house(A(col:num_rows, col));

  % Apply the Householder vector to the remainder of the matrix
  w = beta * v' * A(col:num_rows, col:num_cols);
  A(col:num_rows, col:num_cols) = A(col:num_rows, col:num_cols) - v * w;

  % Zero out the remainder of the column.
  A((col + 1):num_rows, col) = 0;
end  % for col
```

Figure 9: MATLAB code for a Householder QR decomposition ([4])

The function `house(x)` (see Figure 10) computes a vector v and a scalar β such that $P = I - \beta vv^T$ is orthogonal, where $\beta = \frac{2}{v^T v}$, and $Px = \|x\|_2 e_1$ ($e_1 = [1, 0, \ldots, 0]^T$).

```
if (isreal(x))
  n = length(x);
  sigma = x(2:n)' * x(2:n);
  v = [1; x(2:n)];
  if sigma == 0
    beta = 0;
  else
    mu = sqrt(x(1)^2 + sigma);
    if x(1) <= 0
      v(1) = x(1) - mu;
    else
      v(1) = -sigma / (x(1) + mu);
    end  % if
    beta = 2 * v(1) ^ 2 / (sigma + (v(1) ^ 2));
    v = v / v(1);
  end  % if
else
  v = x;
  nx = norm(x);
  v(1) = x(1) + nx;
  beta = 1 / (nx * (nx + x(1)));
end  % if
```

Figure 10: MATLAB code for computing the Householder vector ([4])

In computing the flop count for a real Householder QR decomposition, we consider the application of the Householder reflections only: the flop count for the computation of the Householder vector is ignored, as it is much smaller than the flop count for the application of the Householder reflections.

For the ith iteration, computing $A^T v$ requires $2(m-i)(n-i)$ flop. Computing $w = \beta A^T v$ requires $m-i$ flop, but is ignored as it is a secondary term. Computing vw^T requires $(m-i)(n-i)$ flop. Computing $A - vw^T$ requires another $(m-i)(n-i)$ flop. The total flop count for the ith column is $4(m-i)(n-i)$ flop.

Because we apply the Householder reflection to the portion of the input matrix that is on or to the right of the diagonal (see Figure 11), the length of the columns decrease by one as we operate on successive columns.

$$\begin{bmatrix} a & a & a & a & a \\ 0 & a & a & a & a \\ 0 & 0 & b & b & b \\ 0 & 0 & b & b & b \\ 0 & 0 & b & b & b \\ 0 & 0 & b & b & b \end{bmatrix}$$

Elements a have already been upper triangularized

The Householder reflection is applied to elements b

Figure 11: Portion of input matrix to which apply the Householder reflection

The total flop count for the entire $m \times n$ input matrix can be computed as follows:

$$\text{flop count} = \sum_{i=1}^{n} 4(m-i)(n-i) \tag{52}$$

$$= 4\sum_{i=1}^{n}[mn-(m+n)i+i^2]$$

$$= 4\sum_{i=1}^{n} mn - 4\sum_{i=1}^{n}(m+n)i + 4\sum_{i=1}^{n} i^2$$

$$= 4mn^2 - 4(m+n)\sum_{i=1}^{n} i + 4\sum_{i=1}^{n} i^2$$

$$= 4mn^2 - 4(m+n)\frac{n(n+1)}{2} + 4\frac{n(n+1)(2n+1)}{6}$$

$$= 4mn^2 - 2(m+n)(n^2+n) + \frac{2}{3}(2n^3+3n^2+n)$$

$$= 4mn^2 - 2mn^2 - 2n^3 - 2mn - 2n^2 + \frac{4}{3}n^3 + 2n^2 + \frac{2}{3}n$$

$$= 2mn^2 - \frac{2}{3}n^3 - 2mn + \frac{2}{3}n$$

We then drop the lower order terms to arrive at the canonical estimated flop count for real matrices:

$$\text{flop count} \cong 2mn^2 - \frac{2}{3}n^3 \tag{53}$$

$$= 2n^2\left(m - \frac{n}{3}\right)$$

4.2 COMPLEX HOUSEHOLDER QR DECOMPOSITION

If the input matrix $A \in C^{m \times n}$ is complex, the QR decomposition factors A into a unitary matrix $Q \in C^{m \times m}$ and an upper triangular matrix $R \in C^{m \times n}$ ([4]):

$$A = QR \tag{54}$$

$$Q^H Q = I, QQ^H = I \tag{55}$$

A MATLAB implementation of the complex Householder QR decomposition, which overwrites the input matrix with the upper triangular factor (the Householder QR decomposition does not return the unitary factor), is the same as the implementation for a real Householder QR decomposition, which was given in Figure 9.

In computing the flop count for a complex Householder QR decomposition, we consider the application of the Householder reflections only; the flop count for the computation of the Householder vector is ignored.

To avoid explicitly computing the Householder reflection $I - \beta vv^H$, we use the following implementation:

$$\text{define } w = \beta A^H v \tag{56}$$

$$(I - \beta vv^H)A = A - \beta vv^H A \tag{57}$$

$$= A - v(\beta v^H A)$$

$$= A - vw^H$$

For the ith iteration, computing $A^H v$ requires $8(m-i)(n-i)$ flop. Computing $w = \beta A^H v$ requires $2(m-i)$ flop, but is ignored. Computing vw^H requires $6(m-i)(n-i)$ flop. Computing $A - vw^H$ requires $2(m-i)(n-i)$ flop. The total flop count for the ith column is $16(m-i)(n-i)$ flop, or four times the flop count for the real Householder QR decomposition. The canonical total estimated flop count for the complex Householder QR decomposition is therefore

$$\text{flop count} \cong 8n^2\left(m - \frac{n}{3}\right) \tag{58}$$

5. FORWARD AND BACK SUBSTITUTIONS

5.1 REAL FORWARD AND BACK SUBSTITUTIONS

A forward substitution allows us to solve the lower triangular system $Lx = b$ for $x \in \mathcal{R}^n$ given lower triangular $L \in \mathcal{R}^{n \times n}$ and $b \in \mathcal{R}^n$. Fundamentally, the forward substitution process is as follows ([4]). First, we solve for the first unknown x_1:

$$L_{11}x_1 = b_1 \tag{59}$$

$$x_1 = \frac{b_1}{L_{11}} \tag{60}$$

Next, we use this value of x_1 to solve for x_2:

$$L_{21}x_1 + L_{22}x_2 = b_2 \tag{61}$$

$$L_{22}x_2 = b_2 - L_{21}x_1 \tag{62}$$

$$x_2 = \frac{b_2 - L_{21}x_1}{L_{22}} \tag{63}$$

Continuing forward, we can solve for all elements of the vector x.

A MATLAB implementation of the forward substitution, which overwrites the input vector b with the solution vector x, is shown below in Figure 12.

```
b(1) = b(1) / L(1, 1);
for row = 2:num_rows
  b(row) = (b(row) - L(row, 1:row - 1) * b(1:row - 1)) / L(row, row);
end  % for row
```

Figure 12: MATLAB code for a forward substitution ([4])

In computing the flop count for a forward substitution, we ignore the workload necessary to compute $x_1 = \frac{b_1}{L_{11}}$, as it will be a secondary term.

For row i, computing the dot product $L_{i,\,1:i-1} \times x_{1:i-1}$ requires $2(i-1)$ flop. Subtracting this dot product from b_i requires one flop, but is ignored as it is a secondary term. Multiplying the difference by $1/L_{i,i}$ also requires one flop and is also ignored.

The flop count for all n elements of x can be computed as follows:

$$\begin{aligned}
\text{flop count} &= \sum_{i=1}^{n} 2(i-1) \qquad (64)\\
&= -2n + 2\sum_{i=1}^{n} i\\
&= -2n + 2\frac{n(n+1)}{2}\\
&= -2n + n^2 + n\\
&= n^2 - n
\end{aligned}$$

We then drop the lower order term to arrive at the canonical estimated flop count for a forward substitution on real matrices:

$$\text{flop count} \cong n^2 \qquad (65)$$

A back substitution is the analog of the forward substitution for upper triangular matrices, letting us solve the triangular system $Ux = b$ for $x \in \mathcal{R}^n$ given upper triangular $U \in \mathcal{R}^{n \times n}$ and $b \in \mathcal{R}^n$. Instead of starting with the first unknown and working forward, we start with the last unknown x_n and work back.

A MATLAB implementation of the back substitution, which overwrites the input vector b with the solution vector x, is shown below in Figure 13.

```
b(num_rows) = b(num_rows) / U(num_rows, num_rows);
for row = (num_rows - 1):-1:1
  b(row) = (b(row) - U(row, row + 1:num_rows) * b(row + 1:num_rows)) ...
             / U(row, row);
end  % for row
```

Figure 13: MATLAB code for a back substitution ([4])

The derivation of the flop count for a back substitution is almost identical to the derivation for the forward substitution flop count, giving us the canonical estimated flop count for a back substitution:

$$\text{flop count} \cong n^2 \qquad (66)$$

5.2 COMPLEX FORWARD AND BACK SUBSTITUTIONS

The algorithms used for real forward and back substitutions may be used for complex forward and back substitutions.

In a forward substitution, for row i, computing the dot product $L_{i,\,1:i-1} \times x_{1:i-1}$ requires $8(i-1)$ flop. Subtracting this dot product from b_i requires two flop, but is ignored as it is a secondary term. Multiplying the difference by $1/L_{i,\,i}$ also requires two flop and is also ignored. The flop count for row i is four times the flop count for the corresponding computation for a real forward substitution. The canonical total estimated flop count for a complex forward substitution is therefore

$$\text{flop count} \cong 4n^2 \qquad (67)$$

The derivation of the flop count for a complex back substitution is almost identical to the derivation for the complex forward substitution flop count, giving us the canonical estimated flop count for a back substitution:

$$\text{flop count} \cong 4n^2 \qquad (68)$$

6. EIGENVALUE DECOMPOSITION

6.1 REAL EIGENVALUE DECOMPOSITION

The symmetric Schur decomposition of a real symmetric matrix $A \in \mathcal{R}^{n \times n}$ computes an orthogonal matrix $Q \in \mathcal{R}^{n \times n}$ such that ([4]):

$$Q^T A Q = \Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n) \tag{69}$$

This decomposition of A results in the eigenvectors being the columns of Q and the corresponding eigenvalues being the diagonal elements of Λ ([2]):

From the definition of eigenvectors and eigenvalues:

$$Aq = \lambda q \text{, where } q_i^T q_i = 1 \text{ and } q_i^T q_j = 0 \text{ for } i \neq j \tag{70}$$

$$AQ = \Lambda Q \text{, where } Q^T Q = QQ^T = I \tag{71}$$

$$Q^T A Q = Q^T \Lambda Q \tag{72}$$

$$\text{Let } B = \Lambda Q \tag{73}$$

$$b_{ij} = \sum_{k=1}^{n} \lambda_{ik} q_{kj} \tag{74}$$

$$= \lambda_i q_{ij} \text{ because } \lambda_{ik} = 0 \text{ for } i \neq k$$

$$\text{Let } C = Q^T \Lambda Q = Q^T B \tag{75}$$

$$c_{ij} = \sum_{k=1}^{n} q_{ki} b_{kj} \tag{76}$$

$$= \sum_{k=1}^{n} q_{ki}(\lambda_k q_{kj})$$

$$= \sum_{k=1}^{n} \lambda_k q_{ki} q_{kj}$$

$$= \begin{cases} 0 \text{ if } i \neq j \\ \lambda_i \text{ if } i = j \end{cases}$$

$$\therefore Q^T A Q = \Lambda \tag{77}$$

As the first step in the implementation of the symmetric Schur decomposition, we tridiagonalize the input matrix:

$$\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix} \Rightarrow \begin{bmatrix} y & y & 0 & 0 & 0 \\ y & y & y & 0 & 0 \\ 0 & y & y & y & 0 \\ 0 & 0 & y & y & y \\ 0 & 0 & 0 & y & y \end{bmatrix} \tag{78}$$

The tridiagonal matrix T is derived from the input matrix A through orthogonal Householder reflections Q_T:

$$Q_T{}^T A Q_T = T \tag{79}$$

A MATLAB implementation of the real Householder tridiagonalization, which overwrites the input matrix with the tridiagonal matrix (if Q_T is desired, it must be separately formed), is shown below in Figure 14.

```
for k = 1:(num_rows - 2)
  % Compute the Householder vector v.
  [v, beta] = house(A((k + 1):num_rows, k));

  % Apply the Householder reflection.
  p = beta' * A((k + 1):num_rows, (k + 1):num_rows) * v;
  w = p - (beta * v * v' * p / 2);
  A((k + 1), k) = -norm(A((k + 1):num_rows, k));
  A(k, (k + 1)) = A((k + 1), k);
  A((k + 1):num_rows, (k + 1):num_rows) ...
    = A((k + 1):num_rows, (k + 1):num_rows) - v * w' - w * v';

  % Zero out remainder of row and column.
  A((k + 2):num_rows, k) = 0;
  A(k, (k + 2):num_rows) = 0;
end  % for k

% Apply phase-only correction to last super- and sub-diagonal elements to
% make them real
last_super_diag = A((num_rows - 1), num_rows);
A((num_rows - 1), num_rows) = abs(last_super_diag);
A(num_rows, (num_rows - 1)) = abs(last_super_diag);
```

Figure 14: MATLAB code for a real Householder tridiagonalization ([4])

In computing the flop count for a real Householder tridiagonalization, we consider the application of the Householder reflections only: the flop count for the computation of the Householder vector v is ignored.

For iteration k, computing the product of $A(k+1\text{:num\_rows}, k+1\text{:num\_rows})$ and v requires $2(n-k)^2$ flop. Scaling this product by β to compute p requires $n-k$ flop and is disregarded. Computing w requires $3(n-k)+2$ is also disregarded. Computing $\|A(k+1\text{:num\_rows}, k)\|_2$ requires $2(n-k)$ flop and is also disregarded.

Computing vw^T requires $(n-k)^2$ flop. The product wv^T is the transpose of vw^T and requires no additional computations. Taking advantage of the symmetry of the output, subtracting both vw^T and wv^T from A requires $(n-k)^2$ flop. The total flop count for iteration k is $4(n-k)^2$ flop.

The total flop count for the entire $n \times n$ matrix can be computed as follows:

$$\text{flop count} = \sum_{k=1}^{n-2} 4(n-k)^2 \tag{80}$$

$$= 4 \times \left(\sum_{k=1}^{n-2} n^2 - 2 \sum_{k=1}^{n-2} nk + \sum_{k=1}^{n-2} k^2 \right)$$

$$= 4 \times \left(n^2 \sum_{k=1}^{n-2} 1 - 2n \sum_{k=1}^{n-2} k + \sum_{k=1}^{n-2} k^2 \right)$$

$$= 4 \times \left\{ [n^2(n-2)] - \left[2n \frac{(n-2)(n-1)}{2} \right] + \left[\frac{(n-2)(n-1)(2n-3)}{6} \right] \right\}$$

$$= \frac{8n^3 - 12n^2 + 4n - 24}{6}$$

We then drop the lower order terms to arrive at the canonical estimated flop count for real matrices:

$$\text{flop count} \cong \frac{4}{3} n^3 \tag{81}$$

To evaluate the eigenvectors in addition to the eigenvalues, we will need to accumulate the Householder reflections in Q_T. The product of the Householder reflection matrices is equal to

$$Q_T = P_1 \ldots P_{n-2} \tag{82}$$

where each matrix P_k is the Householder reflection matrix for loop index k in the algorithm given in Figure 14 above. Each matrix P_k has the form

$$P_k = \begin{array}{cc} \begin{bmatrix} I_k & 0 \\ 0 & \bar{P}_k \end{bmatrix} & \begin{array}{c} k \\ n-k \end{array} \\ \begin{array}{cc} k & n-k \end{array} & \end{array} \tag{83}$$

where I_k is the $k \times k$ identity matrix and

$$\bar{P}_k = I_{n-k} - \beta v v^T \tag{84}$$

for iteration k.

Noting that the $\bar{P}_k$ portion of P_k shrinks as k increases, we can reduce the workload needed to compute Q_T if we accumulate this product starting with P_{n-2} (the Householder transformation matrix with the smallest non-identity submatrix) rather than starting with P_1 (the Householder transformation matrix with the largest non-identity submatrix): instead of accumulating a product that is virtually completely non-identity from the beginning, we will slowly grow the submatrix that is non-identity ([6]).

Let

$$Q_k = \prod_{i=k}^{n-2} P_i \tag{85}$$

Then

$$Q_{k-1} = P_{k-1}Q_k \tag{86}$$

and

$$Q_T = Q_1 \tag{87}$$

If we consider the non-identity portion of the product $Q_{k-1} = P_{k-1}Q_k$, we have

$$\bar{Q}_{k-1} = \bar{P}_{k-1}\bar{Q}_k \tag{88}$$

where

$$Q_k = \begin{bmatrix} I & 0 \\ 0 & \bar{Q}_k \end{bmatrix} \tag{89}$$

Given that the Householder reflection matrix $\bar{P}_{k-1}$ is

$$\bar{P}_{k-1} = I - \beta v v^T \tag{90}$$

we can express the product in Equation 88 thusly:

$$\begin{aligned} \bar{Q}_{k-1} &= (I - \beta v v^T)\bar{Q}_k \\ &= \bar{Q}_k - \beta v v^T \bar{Q}_k \end{aligned} \tag{91}$$

Let

$$w = v^T \bar{Q}_k \tag{92}$$

Then

$$\overline{Q}_{k-1} = \overline{Q}_k - \beta v w \tag{93}$$

Computing the product $w = v^T\overline{Q}_k$ requires $2(n-k)^2$ flop. Scaling w by β requires $n-k$ flop and is disregarded. Computing $\beta v w$ requires $(n-k)^2$ flop. Computing $\overline{Q}_k - \beta v w$ also requires $(n-k)^2$, for a total of $4(n-k)^2$ flop for iteration k. The total flop count to accumulate Q_T for a real input matrix is

$$\text{flop count} \cong \frac{4}{3}n^3 \tag{94}$$

The rest of the work in the symmetric Schur decomposition is performed in a series of implicit symmetric QR steps with Wilkinson shifts ([4]). This algorithm takes as an input an unreduced symmetric tridiagonal matrix $T \in \mathcal{R}^{n \times n}$ and overwrites it with the quantity Z^TTZ. A matrix is said to be unreduced if it has no zero subdiagonal entries. The matrix Z is equal to the product of Givens rotations

$$Z = G_1 \ldots G_{n-1} \tag{95}$$

and has the property that $Z^T(T - \mu I)$ is upper triangular. The scalar μ is the eigenvalue of the 2-by-2 principal submatrix of T that is closer to t_{nn} (the element in the matrix T in the nth row and nth column).

There is an easy way to compute the eigenvalues of a 2×2 matrix. First, we observe that the eigenvalues λ of a matrix A satisfy the characteristic polynomial ([4])

$$det(\lambda I - A) = 0 \tag{96}$$

Expanding Equation 96:

$$det\left(\begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} - \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\right) = 0 \tag{97}$$

$$det\left(\begin{bmatrix} \lambda - a_{11} & -a_{12} \\ -a_{21} & \lambda - a_{22} \end{bmatrix}\right) = 0 \tag{98}$$

The determinant of a 2×2 matrix is given by

$$det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc \tag{99}$$

Therefore, Equation 98 is equivalent to

$$(\lambda - a_{11})(\lambda - a_{22}) - (-a_{12})(-a_{21}) = 0 \qquad (100)$$

$$\lambda^2 - (a_{11} + a_{22})\lambda + (a_{11}a_{22} - a_{12}a_{21}) = 0 \qquad (101)$$

Solving for λ, we have:

$$\lambda = \frac{(a_{11} + a_{22}) \pm \sqrt{(a_{11} + a_{22})^2 - 4(a_{11}a_{22} - a_{12}a_{21})}}{2} \qquad (102)$$

The implicit symmetric QR step with Wilkinson shift uses the algorithm `givens`(a, b), which returns two scalars $c = \cos(\theta)$ and $s = \sin(\theta)$ such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \qquad (103)$$

A MATLAB implementation of the `givens` algorithm is shown below in Figure 15.

```
function: [c, s] = givens(a, b)

  if $b = 0$
    $c = 1$
    $s = 0$
  else
    if $|b| > |a|$
      $\tau = -a/b$
      $s = 1/\sqrt{1+\tau^2}$
      $c = s\tau$
    else
      $\tau = -b/a$
      $c = 1/\sqrt{1+\tau^2}$
      $s = c\tau$
    end  % if
  end  % if
```

Figure 15: MATLAB code for the `givens` algorithm ([4])

This algorithm requires five flop and a single square root.

A shorthand notation is often used for a real Givens rotation matrix:

$$G(i, k, \theta) = \begin{array}{cc} \left[\begin{array}{ccccccc} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \dots & & \dots & & \dots & & \dots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \dots & & \dots & & \dots & & \dots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \dots & & \dots & & \dots & & \dots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{array}\right] & \begin{array}{c} \\ \\ i \\ \\ k \\ \\ \\ \end{array} \\ \begin{array}{ccccccc} & & i & & k & & \end{array} & \end{array} \tag{104}$$

When applying a Givens rotation, we do not explicitly form this matrix; instead, we take advantage of the structure in the Givens rotation matrix.

Let $A \in \mathcal{R}^{m \times n}$, $c = \cos(\theta)$, and $s = \sin(\theta)$. Then, the update $A \leftarrow G(i, k, \theta)^T A$ affects just rows i and k of A:

$$A([i, k], :) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T A([i, k], :) \tag{105}$$

A MATLAB implementation of this update is shown below in Figure 16.

```
for $j$ = 1:n
  $\tau_1$ = A(i, j)
  $\tau_2$ = A(k, j)
  A(i, j) = $c\tau_1 - s\tau_2$
  A(k, j) = $s\tau_1 + c\tau_2$
end  % for $j$
```

Figure 16: MATLAB code for pre-applying a Givens rotation ([4])

This update requires only $6n$ flop.

Similarly, the update $A \leftarrow AG(i, k, \theta)$ affects only columns i and k of A:

$$A(:, [i, k]) = A(:, [i, k])\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \tag{106}$$

A MATLAB implementation of this update is shown below in Figure 17.

```
for $j$ = 1:m
  $\tau_1$ = A(j, i)
  $\tau_2$ = A(j, k)
  A(j, i) = $c\tau_1 - s\tau_2$
  A(j, k) = $s\tau_1 + c\tau_2$
end  % for $j$
```

Figure 17: MATLAB code for post-applying a Givens rotation ([4])

This update also requires only $6n$ flop.

A MATLAB implementation of the implicit symmetric QR step with Wilkinson shift is shown below in Figure 18.

```
$d = (t_{n-1,n-1} - t_{nn})/2$
$\mu = t_{nn} - t_{n,n-1}{}^2/(d + \mathrm{sign}(d)\sqrt{d^2 + t_{n,n-1}{}^2})$
$x = t_{11} - \mu$
$z = t_{21}$
for k = 1:n - 1
  [c, s] = givens(x, z);
  $T = G_k{}^T T G_k$, where $G_k = G(k, k+1, \theta)$
  if k < n - 1
    $x = t_{k+1,k}$
    $z = t_{k+2,k}$
  end  % if
end  % for k
```

Figure 18: MATLAB code for an implicit symmetric QR step with Wilkinson shift ([4])

The bulk of the workload is found in the `for` loop. For each `k`, we need to perform:

- five flop and one square root for the `givens` algorithm
- 27 flop to compute $G_k{}^T T G_k$ (see below)

for a total of 32 flop and one square root.

The flop count for applying the Givens rotation G_k to T is only 27 and independent of the matrix size n because we can take advantage of the fact that, within the for k loop, ${G_k}^T T G_k$ remains symmetric, and that most of the elements in rows k and $k+1$ are zero. Consider the premultiplication of T by ${G_k}^T$ (see Figure 19).

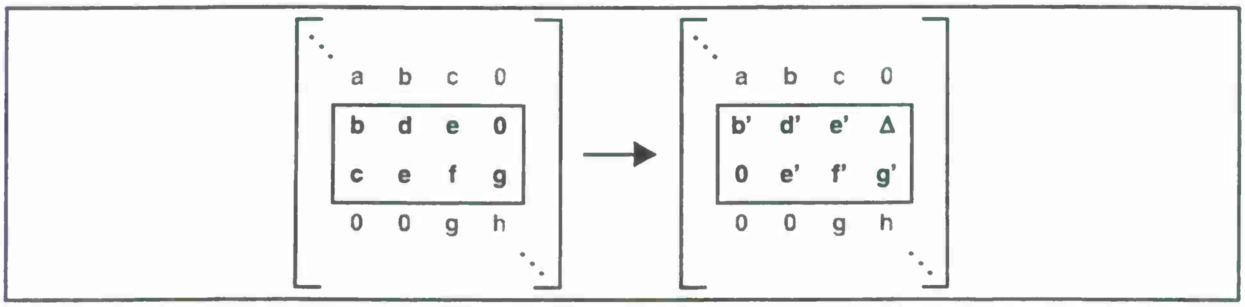

Figure 19: Premultiplication by the Givens rotation matrix

To form the product ${G_k}^T T$, we only need to compute b', d', e', f', g', and Δ, even though we are updating a total of eight entries: we don't need to compute a value we know to be zero, and we only need to compute e' once. Computing these scalars requires three flop per scalar (to compute either $c\tau_1 - s\tau_2$ or $s\tau_1 + c\tau_2$), for a total of 18 flop.

We can similarly take advantage of the structure in the matrices to efficiently postmultiply ${G_k}^T T$ by G_k (see Figure 20).

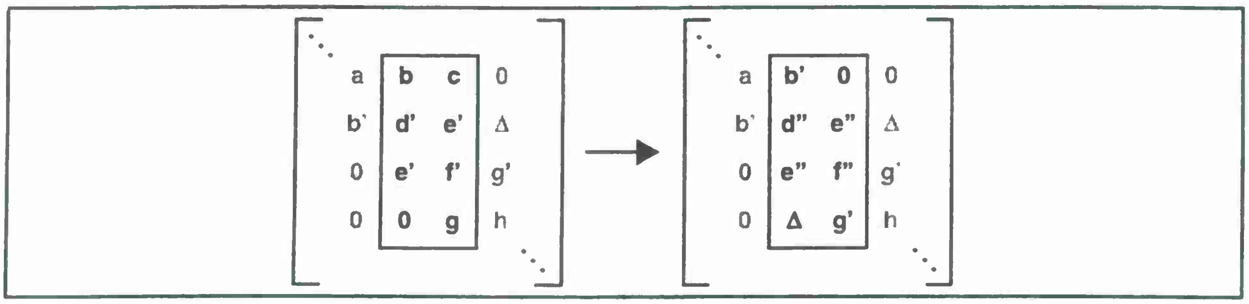

Figure 20: Postmultiplication by the Givens rotation matrix

To compute the product ${G_k}^T T G_k$, we only need to compute d'', e'', and f'', even though we are updating eight entries: we don't need to compute a value we know to be zero, and we can take advantage of the symmetry in ${G_k}^T T G_k$ to avoid recomputing b', e'', g', and Δ. Computing these scalars requires three flop per scalar, for a total of nine flop.

For all `k` ranging from 1 to `n - 1`, we will need to perform $32(n-1)$ flop and $n-1$ square roots, which we round to $30n$ flop and n square roots to match the text ([4]).

If we want to accumulate the Givens rotations by updating an input orthogonal matrix Q with $QG_1 \ldots G_{n-1}$, we will require $6n$ flop to apply each Givens rotation to the running product, for a total of $6n(n-1)$, or approximately $6n^2$ flop.

A MATLAB pseudo-code implementation for the overall algorithm for computing the symmetric Schur decomposition of a real matrix A, which overwrites A with the tridiagonal matrix T, is given below in Figure 21 (`tol` is a tolerance greater than the unit roundoff).

```
% Tridiagonalize the input matrix.
Use the algorithm in Figure 14 to compute the tridiagonalization
  $T = (P_1 \ldots P_{n-2})^T A (P_1 \ldots P_{n-2})$
  Set $D = T$ and, if $Q$ is desired, form $Q = P_1 \ldots P_{n-2}$

until q = n
  for i = 1:n - 1
    if $|d_{i+1,i}| = |d_{i,i+1}| \le \mathrm{tol} \times (|d_{ii}| + |d_{i+1,i+1}|)$
      $d_{i+1,i}$ = 0;
      $d_{i,i+1}$ = 0;
    end  % if
  end  % for i

  Find the largest $q$ and the smallest $p$ such that, if

      $D = \begin{array}{cc} \left[\begin{array}{ccc} D_{11} & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & D_{33} \end{array}\right] & \begin{array}{c} p \\ n-p-q \\ q \end{array} \\ \begin{array}{ccc} p & n-p-q & q \end{array} & \end{array}$

    then $D_{33}$ is diagonal and $D_{22}$ is unreduced

  if $q < n$
    Use the algorithm in Figure 18 to update $D_{22}$:
      $D_{22} = \mathrm{diag}(I_p, \bar{Z}, I_q)^T D_{22}\, \mathrm{diag}(I_p, \bar{Z}, I_q)$
    If $Q$ is desired, then update $Q$:
      $Q = Q\, \mathrm{diag}(I_p, \bar{Z}, I_q)$
  end  % if
end  % until q = n
```

Figure 21: MATLAB pseudo-code for a real symmetric Schur decomposition ([4])

The matrix I_n is the $n \times n$ identity matrix.

The computational workload for the real symmetric Schur decomposition is found principally in the tridiagonalization of the input matrix, which requires approximately $\frac{4}{3}n^3$ flop. There are $O(n)$ calls to the implicit symmetric QR step with Wilkinson Shift, each with an $O(n)$ flop count. Therefore, the flop count for the implicit symmetric QR steps will be $O(n^2)$. As it is an order smaller than the flop count for the tridiagonalization, this flop count is ignored in the overall flop count, giving us the canonical flop count for the real symmetric Schur decomposition:

$$\text{flop count} \cong \frac{4}{3}n^3 \tag{107}$$

It should be noted here that the actual number of iterations through the algorithm in Figure 21 needed to converge on a solution is not deterministic. We estimate this number of iterations to be approximately n, but it may be several times larger than n.

If we want to accumulate the orthogonal transformations Q, we will need to:

- accumulate the Householder reflections during the tridiagonalization of the input matrix, at a cost of $\frac{4}{3}n^3$ flop
- accumulate the Givens rotations during each of the $O(n)$ implicit symmetric QR steps at a cost of $6n^2$ flop per iteration, for a total of approximately $6n^3$ flop

If we add to these two items to the workload for the symmetric Schur algorithm without accumulating Q, we can compute the total workload:

$$\begin{aligned} \text{flop count} &= \frac{4}{3}n^3 + 6n^3 + \frac{4}{3}n^3 \\ &= \frac{26}{3}n^3 \end{aligned} \tag{108}$$

We round this figure to arrive at the canonical workload for the symmetric Schur algorithm if we accumulate Q:

$$\text{flop count} \cong 9n^3 \tag{109}$$

6.2 COMPLEX EIGENVALUE DECOMPOSITION

The symmetric Schur decomposition of a complex symmetric matrix $A \in C^{n \times n}$ computes a unitary matrix $Q \in C^{n \times n}$ such that:

$$Q^H A Q = \Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n) \tag{110}$$

This decomposition of A results in the eigenvectors being the columns of Q and the corresponding eigenvalues being the diagonal elements of Λ.

As the first step in the implementation of the symmetric Schur decomposition, we tridiagonalize the input matrix. The tridiagonal matrix T is derived from the input matrix A through unitary Householder reflections Q_T:

$$Q_T^H A Q_T = T \tag{111}$$

A MATLAB implementation of the complex Householder tridiagonalization, which overwrites the input matrix with the tridiagonal matrix (if Q_T is desired, it must be separately formed), is shown below in Figure 22.

```
for k = 1:num_rows - 2
  % Compute the Householder vector v.
  x = A(col:num_rows, col);
  v =  $x \pm e^{i\theta}\|x\|_2 e_1$ ; where  $x_1 = re^{i\theta}$
  beta = 2 / (v' * v);

  % Apply the Householder reflection.
  p = beta * A(k + 1:n, k + 1:n) * v;
  w = p - (beta * p' * v / 2) * v;
  A(k + 1, k) = norm(A(k + 1:num_rows, k));
  A(k, k + 1) = A(k + 1, k);
  A(k + 1:num_rows, k + 1:num_rows) ...
    = A(k + 1:num_rows, k + 1:num_rows) - v * w' - w * v';
end  % for k
```

Figure 22: MATLAB code for a complex Householder tridiagonalization

Even though the input matrix A is complex, the resultant tridiagonal matrix T is real.

In computing the flop count for a complex Householder tridiagonalization, we consider the application of the Householder reflections only: the flop count for the computation of the Householder vector v is ignored.

For iteration k, computing the product of $A(k+1\text{:num\_rows}, k+1\text{:num\_rows})$ and v requires $8(n-k)^2$ flop. Scaling this product by β to compute p requires $2(n-k)$ flop and is disregarded. Computing w requires $12(n-k)+3$ is also disregarded. Computing $\|A(k+1\text{:num\_rows}, k)\|_2$ requires $4(n-k)$ flop and is also disregarded.

Computing vw^H requires $6(n-k)^2$ flop. The product wv^H is the transpose of vw^H and requires no additional computations. Taking advantage of the symmetry of the output, subtracting both vw^H and wv^H from A requires $2(n-k)^2$ flop. The total flop count for iteration k is $16(n-k)^2$ flop, or four times the flop count for the real Householder tridiagonalization. The total estimated flop count for the complex Householder tridiagonalization is therefore

$$\text{flop count} \cong \frac{16}{3}n^3 \tag{112}$$

To evaluate the eigenvectors in addition to the eigenvalues, we will need to accumulate the Householder reflections in Q_T. The product of the Householder reflection matrices is equal to

$$Q_T = P_1 \dots P_{n-2} \tag{113}$$

where each matrix P_k is the Householder reflection matrix for the loop index k in the algorithm given in Figure 22 above. We use the same technique for accumulating Q_T that was used for the real Householder tridiagonalization. For each iteration k, we are computing the quantity

$$\bar{Q}_{k-1} = \bar{Q}_k - \beta v w \tag{114}$$

where

$$w = v^H \bar{Q}_k \tag{115}$$

Computing the product $w = v^H \bar{Q}_k$ requires $8(n-k)^2$ flop. Scaling w by β requires $2(n-k)$ flop and is disregarded. Computing $\beta v w$ requires $6(n-k)^2$ flop. Computing $\bar{Q}_k - \beta v w$ requires $2(n-k)^2$, for a total of $16(n-k)^2$ flop for iteration k, or four times the flop count for the real Householder tridiagonalization. The total flop count to accumulate Q_T for a complex input matrix is

$$\text{flop count} \cong \frac{16}{3}n^3 \tag{116}$$

The rest of the work in the symmetric Schur decomposition is performed in a series of implicit symmetric QR steps with Wilkinson shifts. This algorithm takes as an input an unreduced symmetric tridiagonal matrix $T \in \mathcal{R}^{n \times n}$ and overwrites it with the quantity $Z^T T Z$. The matrix Z is equal to the product of Givens rotations

$$Z = G_1 \dots G_{n-1} \tag{117}$$

and has the property that $Z^T(T - \mu I)$ is upper triangular. Because the tridiagonal matrix T is real, the Givens rotations G_i are also real, and, therefore, the algorithm for implicit symmetric QR steps with Wilkinson shifts that was given in Figure 18 may be used here ([9]). The workload for this algorithm, as was previously indicated, is approximately $30n$ flop and n square roots.

If we want to accumulate the Givens rotations by updating an input unitary matrix Q with $QG_1 \ldots G_{n-1}$, we will require $6n$ flop to apply each Givens rotation to the running product for each of the real and imaginary halves of Q, for a total of $12n(n-1)$, or approximately $12n^2$ flop.

A MATLAB pseudo-code implementation for the overall algorithm for computing the symmetric Schur decomposition of a complex matrix A, which overwrites A with the tridiagonal matrix T, is given below in Figure 23 (`tol` is a tolerance greater than the unit roundoff).

```
% Tridiagonalize the input matrix.
Use the algorithm in Figure 22 to compute the tridiagonalization
  $T = (P_1 \ldots P_{n-2})^H A(P_1 \ldots P_{n-2})$
  Set $D = T$ and, if $Q$ is desired, form $Q = P_1 \ldots P_{n-2}$

until q = n
  for i = 1:n - 1
    if $|d_{i+1,i}| = |d_{i,i+1}| \le \text{tol} \times (|d_{ii}| + |d_{i+1,i+1}|)$
      $d_{i+1,i}$ = 0;
      $d_{i,i+1}$ = 0;
    end % if
  end % for i

  Find the largest $q$ and the smallest $p$ such that, if

        $D = \begin{array}{cc} \begin{bmatrix} D_{11} & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & D_{33} \end{bmatrix} & \begin{array}{c} p \\ n-p-q \\ q \end{array} \\ \begin{array}{ccc} p & n-p-q & q \end{array} & \end{array}$

    then $D_{33}$ is diagonal and $D_{22}$ is unreduced

  if $q<n$
    Use the algorithm in Figure 18 to update $D_{22}$:
      $D_{22} = \text{diag}(I_p, \bar{Z}, I_q)^T D_{22}\, \text{diag}(I_p, \bar{Z}, I_q)$
    If $Q$ is desired, then update $Q$:
      $Q = Q\, \text{diag}(I_p, \bar{Z}, I_q)$
  end % if
end % until q = n
```

Figure 23: MATLAB pseudo-code for a complex symmetric Schur decomposition

The computational workload for the complex symmetric Schur decomposition is found principally in the tridiagonalization of the input matrix, which requires approximately $\frac{16}{3}n^3$ flop. There are $O(n)$ calls to the implicit symmetric QR step with Wilkinson Shift, each with an $O(n)$ flop count. Therefore, the flop count for the implicit symmetric QR steps will be $O(n^2)$. As it is an order smaller than the flop count for the tridiagonalization, this flop count is ignored in the overall flop count, giving us the flop count for the complex symmetric Schur decomposition:

$$\text{flop count} \cong \frac{16}{3}n^3 \tag{118}$$

If we want to accumulate the orthogonal transformations Q, we will need:

- accumulate the Householder reflections during the tridiagonalization of the input matrix, at a cost of $\frac{16}{3}n^3$ flop
- accumulate the Givens rotations during each of the $O(n)$ implicit symmetric QR steps at a cost of $12n^2$ flop per iteration, for a total of approximately $12n^3$ flop

If we add to these two items the workload for the symmetric Schur algorithm without accumulating Q, we can compute the total workload:

$$\begin{aligned} \text{flop count} &= \frac{16}{3}n^3 + 12n^3 + \frac{16}{3}n^3 \\ &= \frac{68}{3}n^3 \end{aligned} \tag{119}$$

We round this figure to arrive at the workload for the symmetric Schur algorithm if we accumulate Q:

$$\text{flop count} \cong 23n^3 \tag{120}$$

7. SINGULAR VALUE DECOMPOSITION

7.1 REAL SINGULAR VALUE DECOMPOSITION

The singular value decomposition (SVD) of a real matrix $A \in \mathcal{R}^{m \times n}$, where $m \geq n$, computes orthogonal matrices $U \in \mathcal{R}^{m \times m}$ and $V \in \mathcal{R}^{n \times n}$ such that ([4]):

$$U^T A V = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n) \tag{121}$$

where

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0 \tag{122}$$

The scalars σ_i are the singular values of A. The matrix U contains the left singular vectors, while the matrix V contains the right singular vectors.

As the first step in the implementation of the SVD algorithm, we upper bidiagonalize the input matrix:

$$\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix} \rightarrow \begin{bmatrix} y & y & 0 & 0 & 0 \\ 0 & y & y & 0 & 0 \\ 0 & 0 & y & y & 0 \\ 0 & 0 & 0 & y & y \\ 0 & 0 & 0 & 0 & y \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{123}$$

The $n \times n$ bidiagonal matrix B is derived from the input matrix A through the application of orthogonal Householder reflections U_B and V_B:

$$\begin{bmatrix} B \\ 0 \end{bmatrix} = U_B{}^T A V_B \tag{124}$$

A MATLAB implementation of the real Householder bidiagonalization, which overwrites the input matrix with the bidiagonal matrix (if U_B and V_B are desired, they must be separately formed), is shown below in Figure 24.

```
for j = 1:num_cols
  % Compute the Householder vector v.
  [v, beta] = house(A(j:num_rows, j));

  % Apply the Householder reflection (premultiply).
  w = beta * v' * A(j:num_rows, j:num_cols);
  A(j:num_rows, j:num_cols) ...
    = A(j:num_rows, j:num_cols) - v * w;

  % Zero out the rest of the column.
  A((j + 1):num_rows, j) = 0;

  if (j <= (num_cols - 2))
    % Compute the Householder vector v.
    [v, beta] = house((A(j, (j + 1):num_cols))');

    % Apply the Householder reflection (postmultiply).
    w = beta' * A(j:num_rows, (j + 1):num_cols) * v;
    A(j:num_rows, (j + 1):num_cols) ...
      = A(j:num_rows, (j + 1):num_cols) - w * v';

    % Zero out the rest of the row.
    A(j, j + 2:num_cols) = 0;
  end  % if (j <= (num_cols - 2))
end  % for j

% Apply phase-only correction to last super-diagonal element to make it real
last_super_diag = A((num_cols - 1), num_cols);
A((num_cols - 1), num_cols) = abs(last_super_diag);
```

Figure 24: MATLAB code for a Householder bidiagonalization ([4])

In computing the flop count for a real Householder bidiagonalization, we consider the application of the Householder reflections only: the flop count for the computation of the Householder vector v is ignored.

For iteration j, computing the product of the transpose of $A(j\text{:num\_rows}, j\text{:num\_cols})$ and v requires $2(m-j)(n-j)$ flop. Scaling this product by β to compute w requires $m-j$ flop and is disregarded. Computing vw^T requires $(m-j)(n-j)$ flop. Subtracting vw^T from A requires $(m-j)(n-j)$ flop.

If $j \le \text{num\_cols} - 2$, there are additional computations. Computing the product of $A(j\text{:num\_rows}, j+1\text{:num\_cols})$ and v requires $2(m-j)(n-j-1)$. Scaling this product by β requires $n-j-1$ flop and is disregarded. Computing wv^T requires $(m-j)(n-j-1)$. Subtracting wv^T from A requires $(m-j)(n-j-1)$. The total flop count for iteration j is approximately $8(m-j)(n-j)$.

The total flop count for the entire $m \times n$ can be computed as follows:

$$\begin{aligned}
\text{flop count} &= \sum_{j=1}^{n} 8(m-j)(n-j) && (125)\\
&= 8\sum_{j=1}^{n} [mn-(m+n)j+j^2]\\
&= 8\sum_{j=1}^{n} mn - 8\sum_{j=1}^{n}(m+n)j + 8\sum_{j=1}^{n} j^2\\
&= 8mn^2 - 8(m+n)\frac{n(n+1)}{2} + 8\frac{n(n+1)(2n+1)}{6}\\
&= 8mn^2 - 4mn^2 - 4n^3 + \frac{4}{3}(2n^3+3n^2+n)\\
&= 4mn^2 - 4n^3 + \frac{8}{3}n^3 + 4n^2 + \frac{4}{3}n\\
&= 4mn^2 - \frac{4}{3}n^3 + 4n^2 + \frac{4}{3}n
\end{aligned}$$

We then drop the lower order terms to arrive at the canonical estimated flop count for real matrices:

$$\text{flop count} \equiv 4mn^2 - \frac{4}{3}n^3 \qquad (126)$$

To compute the left and right singular vectors, we will need to accumulate the Householder reflections in U_B and V_B. The bidiagonal matrix B is computed from the input matrix A through the following Householder reflections:

$$B = Q_{\text{pre, n}} \cdots Q_{\text{pre, 1}} A Q_{\text{post, 1}} \cdots Q_{\text{post, n-2}} \qquad (127)$$

where $Q_{\text{pre, i}}$ is the Householder reflection matrix (by which we premultiply A) used to zero out column i, and $Q_{\text{post, i}}$ is the Householder reflection matrix (by which we postmultiply A) used to zero out row i, and

n is the number of columns in A. We accumulate all of the $Q_{\text{pre, i}}$ reflection matrices in U_B, and all of the $Q_{\text{post, i}}$ reflection matrices in V_B:

$$U_B^{\;T} = Q_{\text{pre, n}} \cdots Q_{\text{pre, 1}} \text{ or } U_B = Q_{\text{pre, 1}} \cdots Q_{\text{pre, n}} \tag{128}$$

$$V_B = Q_{\text{post, 1}} \cdots Q_{\text{post, n-2}} \tag{129}$$

First, consider the computation of U_B. Each matrix $Q_{\text{pre, j}} \in \mathcal{R}^{m \times m}$ has the form

$$Q_{\text{pre, j}} = \begin{array}{cc} \left[\begin{array}{cc} I_j & 0 \\ 0 & \bar{Q}_{\text{pre, j}} \end{array}\right] & \begin{array}{c} j \\ m-j \end{array} \\ \begin{array}{cc} j & \quad m-j \end{array} & \end{array} \tag{130}$$

where I_j is the $j \times j$ identity matrix and

$$\bar{Q}_{\text{pre, j}} = I_{m-j} - \beta v v^T \tag{131}$$

for iteration j.

Noting that the $\bar{Q}_{\text{pre, j}}$ portion of $Q_{\text{pre, j}}$ shrinks as j increases, we can reduce the workload necessary to compute U_B if we accumulate this product starting with $Q_{\text{pre, n}}$ (the Householder transformation matrix with the smallest non-identity submatrix) rather than starting with $Q_{\text{pre, 1}}$ (the Householder transformation with the largest non-identity submatrix): instead of accumulating a product that is almost completely non-identity from the beginning, we will slowly grow the submatrix that is non-identity.

Let

$$P_{\text{pre, j}} = \prod_{i=j}^{n} Q_{\text{pre, i}} \tag{132}$$

Then

$$P_{\text{pre, j-1}} = Q_{\text{pre, j-1}} P_{\text{pre, j}} \tag{133}$$

and

$$U_B = P_{\text{pre, 1}} \tag{134}$$

If we consider the non-identity portion of the product $P_{\text{pre, j-1}} = Q_{\text{pre, j-1}} P_{\text{pre, j}}$, we have

$$\bar{P}_{\text{pre, j-1}} = \bar{Q}_{\text{pre, j-1}} \bar{P}_{\text{pre, j}} \tag{135}$$

where

$$P_{\text{pre, j}} = \begin{bmatrix} I_j & 0 \\ 0 & \bar{P}_{\text{pre, j}} \end{bmatrix} \tag{136}$$

Given that the Householder reflection matrix $\bar{Q}_{\text{pre, j-1}}$ is

$$\bar{Q}_{\text{pre, j-1}} = I - \beta v v^T \tag{137}$$

we can express the product in Equation 135 thusly:

$$\begin{aligned} \bar{P}_{\text{pre, j-1}} &= (I - \beta v v^T) \bar{P}_{\text{pre, j}} \\ &= \bar{P}_{\text{pre, j}} - \beta v v^T \bar{P}_{\text{pre, j}} \end{aligned} \tag{138}$$

Let

$$w = v^T \bar{P}_{\text{pre, j}} \tag{139}$$

Then

$$\bar{P}_{\text{pre, j-1}} = \bar{P}_{\text{pre, j}} - \beta v w \tag{140}$$

Computing the product $w = v^T \bar{P}_{\text{pre, j}}$ requires $2(m-j)^2$ flop. Scaling w by β requires $m-j$ flop and is disregarded. Computing $\beta v w$ requires $(m-j)^2$ flop. Computing $\bar{P}_{\text{pre, j}} - \beta v w$ also requires $(m-j)^2$ flop, for a total of $4(m-j)^2$ flop for iteration j.

The total flop count to accumulate U_B can be computed as follows:

$$\text{flop count} = \sum_{j=1}^{n} 4(m-j)^2 \tag{141}$$

$$= 4\sum_{j=1}^{n}(m^2 - 2mj + j^2)$$

$$= 4m^2 n - 8m\sum_{j=1}^{n} j + 4\sum_{j=1}^{n} j^2$$

$$= 4m^2 n - 8m\frac{n(n+1)}{2} + 4\frac{n(n+1)(2n+1)}{6}$$

$$= 4m^2 n - 4mn^2 - 4mn + \frac{4}{3}n^3 + 2n^2 + \frac{2}{3}n$$

We then drop the lower order terms to arrive at the estimated flop count for accumulating U_B for real matrices[1]:

$$\text{flop count} \cong 4m^2 n - 4mn^2 + \frac{4}{3}n^3 \tag{142}$$

We can use a similar process to compute V_B. By accumulating the product $V_B = Q_{\text{post, 1}} \cdots Q_{\text{post, n-2}}$ from $Q_{\text{post, n-2}}$ and progressing backward, we can minimize the workload for computing V_B.

Let

$$P_{\text{post, j}} = \prod_{i=j}^{n-2} Q_{\text{post, i}} \tag{143}$$

Then

$$P_{\text{post, j-1}} = Q_{\text{post, j-1}} P_{\text{post, j}} \tag{144}$$

and

$$V_B = P_{\text{post, 1}} \tag{145}$$

If we consider the non-identity portion of the product $P_{\text{post, j-1}} = Q_{\text{post, j-1}} P_{\text{post, j}}$, we have

$$\bar{P}_{\text{post, j-1}} = \bar{Q}_{\text{post. j-1}} \bar{P}_{\text{post, j}} \tag{146}$$

[1] This estimated flop count differs from those in the literature. See Appendix A for more details.

where

$$P_{\text{post, j}} = \begin{bmatrix} I & 0 \\ 0 & \bar{P}_{\text{post, j}} \end{bmatrix} \tag{147}$$

Given that the Householder reflection matrix $\bar{Q}_{\text{post, j-1}}$ is

$$\bar{Q}_{\text{post, j-1}} = I - \beta v v^T \tag{148}$$

we can express the product in Equation 146 thusly:

$$\begin{aligned} \bar{P}_{\text{post, j-1}} &= (I - \beta v v^T)\bar{P}_{\text{post, j}} \\ &= \bar{P}_{\text{post, j}} - \beta v v^T \bar{P}_{\text{post, j}} \end{aligned} \tag{149}$$

Let

$$w = v^T \bar{P}_{\text{post, j}} \tag{150}$$

Then

$$\bar{P}_{\text{post, j-1}} = \bar{P}_{\text{post, j}} - \beta v w \tag{151}$$

Computing the product $w = v^T \bar{P}_{\text{post, j}}$ requires $2(n-j)^2$ flop. Scaling w by β requires $n-j$ flop and is disregarded. Computing $\beta v w$ requires $(n-j)^2$ flop. Computing $\bar{P}_{\text{post, j}} - \beta v w$ also requires $(n-j)^2$ flop, for a total of $4(n-j)^2$ flop for iteration j.

The total flop count to accumulate V_B can be computed as follows:

$$
\begin{aligned}
\text{flop count} &= \sum_{j=1}^{n-2} 4(n-j)^2 \qquad (152)\\
&= 4\sum_{j=1}^{n-2}(n^2-2nj+j^2)\\
&= 4n^2(n-2)-8n\sum_{j=1}^{n-2} j+4\sum_{j=1}^{n-2} j^2\\
&= 4n^3-8n^2-8n\frac{(n-2)(n-1)}{2}+4\frac{(n-2)(n-1)(2n-3)}{6}\\
&= 4n^3-8n^2-(4n^3-12n^2+8n)+\frac{2}{3}(2n^3-9n^2+13n-6)\\
&= \frac{4}{3}n^3-2n^2+\frac{2}{3}n-4
\end{aligned}
$$

We then drop the lower order terms to arrive at the canonical estimated flop count for accumulating V_B for real matrices:

$$\text{flop count} \cong \frac{4}{3}n^3 \qquad (153)$$

The rest of the work in the SVD is performed in a series of Golub-Kahan SVD steps ([4]). This algorithm takes as an input a bidiagonal matrix $B \in \mathcal{R}^{m \times n}$ that has no zeros on its diagonal or superdiagonal[2] and overwrites it with the bidiagonal matrix $\overline{B} = \overline{U}^T B \overline{V}$, where $\overline{U}$ and $\overline{V}$ are orthogonal. A MATLAB pseudo-code implementation of the Golub-Kahan SVD step is shown below in Figure 25.

[2] The diagonal elements of a matrix are those elements whose row and column indices are equal; the superdiagonal elements of a matrix are those elements whose column index is exactly one greater than its row index.

```
Let $\mu$ be the eigenvalue of the trailing 2-by-2 submatrix of $T = B^T B$ that
    is closer to $t_{nn}$
$y = t_{11} - \mu$
$z = t_{12}$
for k = 1:n - 1
  Determine c = $\cos(\theta)$ and s = $\sin(\theta)$ such that $\begin{bmatrix} y & z \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} * & 0 \end{bmatrix}$
  $B \leftarrow BG(k, k+1, \theta)$
  $y = b_{kk}$
  $z = b_{k+1,k}$
  Determine c = $\cos(\theta)$ and s = $\sin(\theta)$ such that $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}$
  $B \leftarrow \hat{G}(k, k+1, \theta)^T B$
  if (k < n - 1)
    $y = b_{k,k+1}$
    $z = b_{k,k+2}$
  end % if
end % for k
```

Figure 25: MATLAB pseudo-code for the Golub-Kahan SVD step ([4])

We can determine $c = \cos(\theta)$ and $s = \sin(\theta)$ by using the givens algorithm shown in Figure 15. Also, we can update B with either $BG(k, k+1, \theta)$ or $G(k, k+1, \theta)^T B$ without explicitly forming the rotation matrix by using the algorithms shown in Figure 17 or Figure 16, respectively.

The bulk of the workload is found in the for loop. For each k, we need to perform:

- ten flop and two square roots for two calls to the givens algorithm
- 12 flop to postmultiply B by $G(k, k+1, \theta)$
- 12 flop to premultiply B by $G(k, k+1, \theta)^T$

for a total of 34 flop and two square roots.

For all k ranging from 1 to n - 1, we will need to perform $34(n-1)$ flop and $2(n-1)$ square roots, which we round to $30n$ flop and $2n$ square roots to match the text [4].

If we want to accumulate the Givens rotations by updating input orthogonal matrices U and V with

$U\hat{G}_1 \ldots \hat{G}_{n-1}$ and $VG_1 \ldots G_{n-1}$, respectively, we will require $6m$ flop to apply each Givens rotation to U and $6n$ flop to apply each Givens rotation to V. For all $n-1$ iterations, we will require $6m(n-1)$ flop, or approximately $6mn$ flop, to accumulate U, and $6n(n-1)$ flop, or approximately $6n^2$ flop, to accumulate V.

A MATLAB pseudo-code implementation of the overall algorithm for computing the SVD of a real matrix A, which overwrites A with its singular values $U^T AV = D + E$, where E satisfies $\|E\|_2 \approx u\|A\|_2$ and u is the unit roundoff, is given below in Figure 26 (ε is a small multiple of the unit roundoff).

```
% Bidiagonalize the input matrix.
Use the algorithm in Figure 24 to compute the bidiagonalization
  $\begin{bmatrix} B \\ 0 \end{bmatrix} = (U_1 \ldots U_n)^T A(V_1 \ldots V_{n-2})$

until q = n
  set $b_{i,i+1}$ to zero if $|b_{i,i+1}| \le \varepsilon(|b_{ii}| + |b_{i+1,i+1}|)$ for any $i \in (1, n-1)$

  find the largest $q$ and the smallest $p$ such that, if

     $B = \begin{array}{cccc} \begin{bmatrix} B_{11} & 0 & 0 \\ 0 & B_{22} & 0 \\ 0 & 0 & B_{33} \end{bmatrix} & \begin{array}{c} p \\ n-p-q \\ q \end{array} \\ \begin{array}{ccc} p & n-p-q & q \end{array} & \end{array}$

    then $B_{33}$ is diagonal and $B_{22}$ has a non-zero superdiagonal

  if q < n
    if any diagonal entry in $B_{22}$ is zero
      zero the superdiagonal entry in the same row
    else
      apply the algorithm in Figure 25 to $B_{22}$
      $B \leftarrow \mathrm{diag}(I_p, U, I_{q+m-n})^T B\mathrm{diag}(I_p, V, I_q)$
    end % if any diagonal entry in $B_{22}$ is zero
  end % if q < n
end % until q = n
```

Figure 26: MATLAB pseudo-code for a real SVD ([4])

The computational workload for the real SVD is found principally in the bidiagonalization of the input matrix, which requires approximately $4mn^2 - \frac{4}{3}n^3$ flop. There are approximately $2n$ calls to the Golub-Kahan SVD step ([3]), each with a $30n$ flop count. Therefore, the flop count for the Golub-Kahan SVD steps will be $60n^2$. As it is an order smaller than the flop count for the bidiagonalization, this flop count is ignored in the overall flop count, giving us the canonical flop count for the real SVD:

$$\text{flop count} \equiv 4mn^2 - \frac{4}{3}n^3 \tag{154}$$

It should be noted here that the actual number of iterations through the algorithm in Figure 26 needed to converge on a solution is not deterministic. We estimate this number of iterations to be approximately $2n$, but it may be quite different, depending on the input matrix and ε.

If we want to accumulate the left singular vectors U, we will need to:

- accumulate U_B during the bidiagonalization of the input matrix, at a cost of $4m^2n - 4mn^2 + \frac{4}{3}n^3$
- accumulate the Givens rotations during each of the $O(n)$ Golub-Kahan SVD steps at a cost of $6mn$ flop per iteration. If we assume that we will require $2n$ Golub-Kahan SVD steps, accumulating the Givens rotations will require a total of approximately $12mn^2$ flop[3]

If we add these two items to the workload for the SVD algorithm without accumulating U, we can compute the canonical expression for the total workload:

$$\begin{aligned}\text{flop count} &= 4mn^2 - \frac{4}{3}n^3 + 4m^2n - 4mn^2 + \frac{4}{3}n^3 + 12mn^2 \\ &= 4m^2n + 12mn^2\end{aligned} \tag{155}$$

Similarly, if we want to accumulate the right singular vectors V, we will need to:

- accumulate V_B during the bidiagonalization of the input matrix, at a cost of $\frac{4}{3}n^3$
- accumulate the Givens rotations during each of the $O(n)$ Golub-Kahan SVD steps at a cost of $6n^2$ flop per iteration. If we assume that we will require $2n$ Golub-Kahan SVD steps, accumulating the Givens rotations will require a total of approximately $12n^3$ flop[4]

[3] This estimated flop count differs from those in the literature. See Appendix A for more details.

[4] This estimated flop count differs from those in the literature. See Appendix A for more details.

If we add these two items to the workload for the SVD algorithm without accumulating V, we can compute the canonical expression for the total workload:

$$\text{flop count} = 4mn^2 - \frac{4}{3}n^3 + \frac{4}{3}n^3 + 12n^3 \qquad (156)$$

$$= 4mn^2 + 12n^3$$

Finally, we can determine the canonical expression for the workload for the SVD algorithm including accumulating both U and V:

$$\text{flop count} = 4mn^2 - \frac{4}{3}n^3 + 4m^2n - 4mn^2 + \frac{4}{3}n^3 + 12mn^2 + \frac{4}{3}n^3 + 12n^3 \qquad (157)$$

$$= 4m^2n + 12mn^2 + \frac{40}{3}n^3$$

$$\cong 4m^2n + 12mn^2 + 13n^3$$

7.2 COMPLEX SINGULAR VALUE DECOMPOSITION

The SVD of a complex matrix $A \in C^{m \times n}$, where $m \geq n$, computes unitary matrices $U \in C^{m \times m}$ and $V \in C^{n \times n}$ such that:

$$U^H A V = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n) \qquad (158)$$

where

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0 \qquad (159)$$

The scalars σ_i are the singular values of A. The matrix U contains the left singular vectors, while the matrix V contains the right singular vectors.

As the first step in the implementation of the SVD algorithm, we upper bidiagonalize the input matrix. The bidiagonal matrix B is derived from the input matrix A through the application of unitary Householder reflections U_B and V_B:

$$\begin{bmatrix} B \\ 0 \end{bmatrix} = U_B{}^H A V_B \qquad (160)$$

A MATLAB implementation of the complex Householder bidiagonalization, which overwrites the input matrix with the bidiagonal matrix (if U_B and V_B are desired, they must be separately formed), is the same as the implementation for a real Householder bidiagonalization, which was given in Figure 24. Even though the input matrix A is complex, the resultant bidiagonal matrix B is real.

In computing the flop count for a complex Householder bidiagonalization, we consider the application of the Householder reflections only: the flop count for the computation of the Householder vector v is ignored.

For iteration j, computing the product of the transpose of $A(j\text{:num\_rows}, j\text{:num\_cols})$ and v requires $8(m-j)(n-j)$ flop. Scaling this product by β to compute w requires $2(m-j)$ flop and is disregarded. Computing vw^T requires $6(m-j)(n-j)$ flop. Subtracting vw^T from A requires $2(m-j)(n-j)$ flop.

If $j \le \text{num\_cols} - 2$, there are additional computations. Computing the product of $A(j\text{:num\_rows}, j+1\text{:num\_cols})$ and v requires $8(m-j)(n-j-1)$. Scaling this product by β requires $2(n-j-1)$ flop and is disregarded. Computing wv^T requires $6(m-j)(n-j-1)$. Subtracting wv^T from A requires $2(m-j)(n-j-1)$. The total flop count for iteration j is approximately $32(m-j)(n-j)$, or four times the flop count for the real Householder bidiagonalization. The total estimated flop count for the complex Householder bidiagonalization is therefore

$$\text{flop count} \cong 16mn^2 - \frac{16}{3}n^3 \tag{161}$$

To compute the left and right singular vectors, we will need to accumulate the Householder reflections in U_B and V_B. The bidiagonal matrix B is computed from the input matrix A through the following Householder reflections:

$$B = Q_{\text{pre, n}} \cdots Q_{\text{pre, 1}} A Q_{\text{post, 1}} \cdots Q_{\text{post, n-2}} \tag{162}$$

where $Q_{\text{pre, i}}$ is the Householder reflection matrix (by which we premultiply A) used to zero out column i, and $Q_{\text{post, i}}$ is the Householder reflection matrix (by which we postmultiply A) used to zero out row i, and n is the number of columns in A. We accumulate all of the $Q_{\text{pre, i}}$ reflection matrices in U_B, and all of the $Q_{\text{post, i}}$ reflection matrices in V_B:

$$U_B^H = Q_{\text{pre, n}} \cdots Q_{\text{pre, 1}} \text{ or } U_B = Q_{\text{pre, 1}} \cdots Q_{\text{pre, n}} \tag{163}$$

$$V_B = Q_{\text{post, 1}} \cdots Q_{\text{post, n-2}} \tag{164}$$

Each matrix $Q_{\text{pre, j}}$ has the form

$$Q_{\text{pre, j}} = \begin{array}{cc} \begin{bmatrix} I_j & 0 \\ 0 & \bar{Q}_{\text{pre, j}} \end{bmatrix} & \begin{array}{c} j \\ m-j \end{array} \\ \begin{array}{cc} j & \quad m-j \end{array} & \end{array} \tag{165}$$

where I_j is the $j \times j$ identity matrix and

$$\bar{Q}_{\text{pre, j}} = I_{m-j} - \beta v v^H \tag{166}$$

for iteration j.

Noting that the $\bar{Q}_{\text{pre, j}}$ portion of $Q_{\text{pre, j}}$ shrinks as j increases, we can reduce the workload necessary to compute U_B if we accumulate this product starting with $Q_{\text{pre, n}}$ (the Householder transformation matrix with the smallest non-identity submatrix) rather than starting with $Q_{\text{pre, 1}}$ (the Householder transformation with the largest non-identity submatrix): instead of accumulating a product that is virtually completely non-identity from the beginning, we will slowly grow the submatrix that is non-identity.

Let

$$P_{\text{pre, j}} = \prod_{i=j}^{n} Q_{\text{pre, i}} \tag{167}$$

Then

$$P_{\text{pre, j-1}} = Q_{\text{pre, j-1}} P_{\text{pre, j}} \tag{168}$$

and

$$U_B = P_{\text{pre, 1}} \tag{169}$$

If we consider the non-identity portion of the product $P_{\text{pre, j-1}} = Q_{\text{pre, j-1}} P_{\text{pre, j}}$, we have

$$\bar{P}_{\text{pre, j-1}} = \bar{Q}_{\text{pre, j-1}} \bar{P}_{\text{pre, j}} \tag{170}$$

where

$$P_{\text{pre, j}} = \begin{bmatrix} I & 0 \\ 0 & \bar{P}_{\text{pre, j}} \end{bmatrix} \tag{171}$$

Given that the Householder reflection matrix $\bar{Q}_{\text{pre, j-1}}$ is

$$\bar{Q}_{\text{pre, j-1}} = I - \beta v v^H \tag{172}$$

we can express the product in Equation 170 thusly:

$$\begin{aligned}\bar{P}_{\text{pre, j-1}} &= (I - \beta v v^H)\bar{P}_{\text{pre, j}} \\ &= \bar{P}_{\text{pre, j}} - \beta v v^H \bar{P}_{\text{pre, j}}\end{aligned} \tag{173}$$

Let

$$w = v^H \bar{P}_{\text{pre, j}} \tag{174}$$

Then

$$\bar{P}_{\text{pre, j-1}} = \bar{P}_{\text{pre, j}} - \beta v w \tag{175}$$

Computing the product $w = v^H \bar{P}_{\text{pre, j}}$ requires $8(m-j)^2$ flop. Scaling w by β requires $2(m-j)$ flop and is disregarded. Computing $\beta v w$ requires $6(m-j)^2$ flop. Computing $\bar{P}_{\text{pre, j}} - \beta v w$ requires $2(m-j)^2$ flop, for a total of $16(m-j)^2$ flop for iteration j, or four times the flop count for the real Householder bidiagonalization. The estimated flop count for accumulating U_B for complex matrices is therefore

$$\text{flop count} \cong 16m^2 n - 16mn^2 + \frac{16}{3}n^3 \tag{176}$$

By accumulating the product $V_B = Q_{\text{post, 1}} \cdots Q_{\text{post, n-2}}$ from $Q_{\text{post, n-2}}$ and progressing backward, we can minimize the workload for computing V_B.

Let

$$P_{\text{post, j}} = \prod_{i=j}^{n-2} Q_{\text{post, i}} \tag{177}$$

Then

$$P_{\text{post, j-1}} = Q_{\text{post. j-1}} P_{\text{post, j}} \tag{178}$$

and

$$V_B = P_{\text{post, 1}} \tag{179}$$

If we consider the non-identity portion of the product $P_{\text{post, j-1}} = Q_{\text{post. j-1}} P_{\text{post, j}}$, we have

$$\bar{P}_{\text{post, j-1}} = \bar{Q}_{\text{post. j-1}}\bar{P}_{\text{post, j}} \tag{180}$$

where

$$P_{\text{post. j}} = \begin{bmatrix} I & 0 \\ 0 & \bar{P}_{\text{post. j}} \end{bmatrix} \tag{181}$$

Given that the Householder reflection matrix $\bar{Q}_{\text{post. j-1}}$ is

$$\bar{Q}_{\text{post. j-1}} = I - \beta v v^H \tag{182}$$

we can express the product in Equation 180 thusly:

$$\begin{aligned} \bar{P}_{\text{post. j-1}} &= (I - \beta v v^H)\bar{P}_{\text{post. j}} \\ &= \bar{P}_{\text{post. j}} - \beta v v^H \bar{P}_{\text{post. j}} \end{aligned} \tag{183}$$

Let

$$w = v^H \bar{P}_{\text{post. j}} \tag{184}$$

Then

$$\bar{P}_{\text{post. j-1}} = \bar{P}_{\text{post. j}} - \beta v w \tag{185}$$

Computing the product $w = v^H \bar{P}_{\text{post. j}}$ requires $8(n-j)^2$ flop. Scaling w by β requires $2(n-j)$ flop and is disregarded. Computing $\beta v w$ requires $6(n-j)^2$ flop. Computing $\bar{P}_{\text{post. j}} - \beta v w$ requires $2(n-j)^2$ flop, for a total of $16(n-j)^2$ flop for iteration j, or four times the flop count for the real Householder bidiagonalization. The total flop count to accumulate V_B for complex matrices is therefore

$$\text{flop count} \cong \frac{16}{3}n^3 \tag{186}$$

The rest of the work in the SVD is performed in a series of Golub-Kahan SVD steps. This algorithm takes as an input a bidiagonal matrix $B \in \mathcal{R}^{m \times n}$ that has no zeros on its diagonal or superdiagonal and overwrites it with the bidiagonal matrix $\bar{B} = \bar{U}^T B \bar{V}$, where $\bar{U}$ and $\bar{V}$ are orthogonal. Because the bidiagonal matrix B is real, the Givens rotations G are also real, and, therefore, the algorithm for the Golub-Kahan SVD steps that was given in Figure 25 may be used here ([9]). The workload for this algorithm, as was previously indicated, is approximately $30n$ flop and $2n$ square roots.

If we want to accumulate the Givens rotations by updating input unitary matrices U and V with

$U\hat{G}_1 \dots \hat{G}_{n-1}$ and $VG_1 \dots G_{n-1}$, respectively, we will require $6m$ flop to apply each Givens rotation to each of the real and imaginary halves of U, and $6n$ flop to apply each Givens rotation to the real and imaginary halves of V. For all $n-1$ iterations, we will require $12m(n-1)$ flop, or approximately $12mn$ flop, to accumulate U, and $12n(n-1)$ flop, or approximately $12n^2$ flop, to accumulate V.

A MATLAB pseudo-code implementation of the overall algorithm for computing the SVD of a complex matrix A, which overwrites A with its singular values $U^H AV = D + E$, is given below in Figure 27 (ε is a small multiple of the unit roundoff).

```
% Bidiagonalize the input matrix.
Use the algorithm in Figure 24 to compute the bidiagonalization
   $\begin{bmatrix} B \\ 0 \end{bmatrix} = (U_1 \dots U_n)^H A(V_1 \dots V_{n-2})$

until q = n
  set $b_{i,i+1}$ to zero if $|b_{i,i+1}| \le \varepsilon(|b_{ii}| + |b_{i+1,i+1}|)$ for any $i \in (1, n-1)$

  find the largest $q$ and the smallest $p$ such that, if

     $B = \begin{array}{cc} \begin{bmatrix} B_{11} & 0 & 0 \\ 0 & B_{22} & 0 \\ 0 & 0 & B_{33} \end{bmatrix} & \begin{array}{c} p \\ n-p-q \\ q \end{array} \\ \begin{array}{ccc} p & n-p-q & q \end{array} & \end{array}$

    then $B_{33}$ is diagonal and $B_{22}$ has a non-zero superdiagonal

  if q < n
    if any diagonal entry in $B_{22}$ is zero
      zero the superdiagonal entry in the same row
    else
      apply the algorithm in Figure 25 to $B_{22}$
      $B \leftarrow \mathrm{diag}(I_p, U, I_{q+m-n})^H B \mathrm{diag}(I_p, V, I_q)$
    end  % if any diagonal entry in $B_{22}$ is zero
  end  % if q < n
end  % until q = n
```

Figure 27: MATLAB pseudo-code for a complex SVD

The computational workload for the complex SVD is found principally in the bidiagonalization of the input matrix, which requires approximately $16mn^2 - \frac{16}{3}n^3$ flop. There are approximately $2n$ calls to the Golub-Kahan SVD step ([3]), each with a $30n$ flop count. Therefore, the flop count for the Golub-Kahan SVD steps will be $60n^2$. As it is an order smaller than the flop count for the bidiagonalization, this flop count is ignored in the overall flop count, giving us the flop count for the complex SVD:

$$\text{flop count} \equiv 16mn^2 - \frac{16}{3}n^3 \qquad (187)$$

If we want to accumulate the left singular vectors U, we will need to:

- accumulate U_B during the bidiagonalization of the input matrix, at a cost of $16m^2n - 16mn^2 + \frac{16}{3}n^3$
- accumulate the Givens rotations during each of the $O(n)$ Golub-Kahan SVD steps at a cost of $12mn$ flop per iteration. If we assume that we will require $2n$ Golub-Kahan SVD steps, accumulating the Givens rotations will require a total of approximately $24mn^2$ flop

If we add these two items to the workload for the SVD algorithm without accumulating U, we can compute the total workload:

$$\begin{aligned}\text{flop count} &= 16mn^2 - \frac{16}{3}n^3 + 16m^2n - 16mn^2 + \frac{16}{3}n^3 + 24mn^2 \qquad (188)\\ &= 16m^2n + 24mn^2\end{aligned}$$

Similarly, if we want to accumulate the right singular vectors V, we will need to:

- accumulate V_B during the bidiagonalization of the input matrix, at a cost of $\frac{16}{3}n^3$
- accumulate the Givens rotations during each of the $O(n)$ Golub-Kahan SVD steps at a cost of $12n^2$ flop per iteration. If we assume that we will require $2n$ Golub-Kahan SVD steps, accumulating the Givens rotations will require a total of approximately $24n^3$ flop

If we add these two items to the workload for the SVD algorithm without accumulating V, we can compute the total workload:

$$\text{flop count} = 16mn^2 - \frac{16}{3}n^3 + \frac{16}{3}n^3 + 24n^3 \tag{189}$$

$$= 16mn^2 + 24n^3$$

Finally, we can determine the workload for the SVD algorithm including accumulating both U and V:

$$\text{flop count} = 16mn^2 - \frac{16}{3}n^3 + 16m^2n - 16mn^2 + \frac{16}{3}n^3 + 24mn^2 + \frac{16}{3}n^3 + 24n^3 \tag{190}$$

$$= 16m^2n + 24mn^2 + \frac{88}{3}n^3$$

$$\cong 16m^2n + 16mn^2 + 29n^3$$

8. SUMMARY

The workload expressions for the real and complex signal processing kernels are summarized below.

Table 1: Flop Count Summary

Signal Processing Kernel	**Computational Complexity**	
	Real Input	**Complex Input**
matrix-matrix multiplication	$2mnp$	$8mnp$
fast Fourier transform	$\frac{5}{2}n\log_2 n$	$5n\log_2 n$
Householder QR decomposition	$2n^2\left(m-\frac{n}{3}\right)$	$8n^2\left(m-\frac{n}{3}\right)$
forward or back substitution	n^2	$4n^2$
eigenvalue decomposition: eigenvalues only	$\frac{4}{3}n^3$	$\frac{16}{3}n^3$
eigenvalue decomposition: eigenvalues and eigenvectors	$9n^3$	$23n^3$
singular value decomposition: singular values only	$4mn^2-\frac{4}{3}n^3$	$16mn^2-\frac{16}{3}n^3$
singular value decomposition: singular values and left singular vectors	$4m^2n+12mn^2$	$16m^2n+24mn^2$
singular value decomposition: singular values and right singular vectors	$4mn^2+12n^3$	$16mn^2+24n^3$
singular value decomposition: singular values, left and right singular vectors	$4m^2n+12mn^2+13n^3$	$16m^2n+16mn^2+29n^3$

In the table above, the parameters in the computational complexity expressions are:

- the dimensions of the two multiplicands - $m \times n$ and $n \times p$ - for the matrix-matrix multiplication

- the length of the vector n for the fast Fourier transform
- the size of the triangular system n for forward and back substitutions
- the dimensions of the input matrix $m \times n$ for the Householder QR decomposition, eigenvalue decomposition, and singular value decomposition.

APPENDIX A. DIFFERENCES IN FLOP COUNTS

Some of the flop counts given in this document differ from those found in the literature. These differences are discussed in greater detail here.

A.1 BIDIAGONALIZATION IN THE SVD

In Section 7.1, we estimated flop count for accumulating U_B for the bidiagonalization of a real matrix to be

$$\text{flop count} \cong 4m^2n - 4mn^2 + \frac{4}{3}n^3 \tag{191}$$

Golub and Van Loan ([4]) estimate this flop count to be $4m^2n - \frac{4}{3}n^3$. This expression would be the estimated flop count if the summation given in Equation 125 was instead

$$\begin{aligned} \text{flop count} &= \sum_{j=1}^{n} 4(m^2 - j^2) \\ &= 4m^2n - 4\sum_{j=1}^{n} j^2 \\ &= 4m^2n - 4\frac{n(n+1)(2n+1)}{6} \\ &= 4m^2n - \left(\frac{4}{3}n^3 + 2n^2 + \frac{2}{3}n\right) \end{aligned} \tag{192}$$

which would then be rounded off to $4m^2n - \frac{4}{3}n^3$.

Chan ([1]) estimates this workload to be $mn^2 - \frac{n^3}{3}$ multiplies. If we assume that there is one addition operation for every multiplication operation, this workload would be equivalent to $2mn^2 - \frac{2n^3}{3}$ flop.

The different flop counts for accumulating U_B for the bidiagonalization of a real matrix are summarized below in Table 2.

Table 2: Workload to Accumulate U_B During the Bidiagonalization of a Real Matrix

Source	Flop Count
Arakawa	$4m^2n - 4mn^2 + \frac{4}{3}n^3$
Chan	$2mn^2 - \frac{2}{3}n^3$
Golub and Van Loan	$4m^2n - \frac{4}{3}n^3$

A.2 ACCUMULATION OF THE LEFT SINGULAR VECTORS IN THE SVD

In Section 7.1, we estimated the flop count for accumulating U during the Golub-Kahan steps for the SVD of a real matrix to be

$$\text{flop count} = 12mn^2 \qquad (193)$$

Golub and Van Loan ([4]) estimate this flop count to be $4mn^2 + \frac{8}{3}n^3$.

Chan ([1]) estimates this workload to be $2mn^2$ multiplies. If we assume that there is one addition operation for every multiplication operation, this workload would be equivalent to $4mn^2$ flop.

The different flop counts for accumulating U during the Golub-Kahan steps for the SVD of a real matrix are summarized below in Table 3.

Table 3: Workload to Accumulate U During the SVD of a Real Matrix

Source	Flop Count
Arakawa	$12mn^2$
Chan	$4mn^2$
Golub and Van Loan	$4mn^2 + \frac{8}{3}n^3$

A.3 ACCUMULATION OF THE RIGHT SINGULAR VECTORS IN THE SVD

In Section 7.1, we estimated the flop count for accumulating V during the Golub-Kahan steps for the SVD of a real matrix to be

$$\text{flop count} = 12n^3 \tag{194}$$

Golub and Van Loan ([4]) estimate this flop count to be $8n^3$.

Chan ([1]) estimates this workload to be $2n^3$ multiplies. If we assume that there is one addition operation for every multiplication operation, this workload would be equivalent to $4n^3$ flop.

The different flop counts for accumulating V during the Golub-Kahan steps for the SVD of a real matrix are summarized below in Table 4.

Table 4: Workload to Accumulate *V* During the SVD of a Real Matrix

Source	Flop Count
Arakawa	$12n^3$
Chan	$4n^3$
Golub and Van Loan	$8n^3$

REFERENCES

[1] T. F. Chan, "An Improved Algorithm for Computing the Singular Value Decomposition", ACM Transactions on Mathematical Software, Vol. 8, 1982, pp. 72-83

[2] D. F. Drake and N. B. Pulsone, private communication, 22 August 2001

[3] G. H. Golub and C. Reinsch, "Singular Value Decomposition and Least Squares Solutions", *Numerische Mathematik*, Vol. 14, 1970, pp. 403-420

[4] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd edition, Baltimore and London: The Johns Hopkins University Press (1996)

[5] J. M. Lebak, private communication, 1 October 2001

[6] R. S. Martin, C. Reinsch, and J. H. Wilkinson, "Householder's Tridiagonalization of a Symmetric Matrix", *Numerische Mathematik*, Vol. 11, 1968, pp. 181-195

[7] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, New Jersey: Prentice Hall (1989)

[8] C. M. Rader, private communication, 2001

[9] C. M. Rader, private communication, 9 April 2002

[10] C. M. Rader, private communication, 11 April 2002

[11] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, Philadelphia: Society for Industrial and Applied Mathematics (1992)

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
28 May 2003

2. REPORT TYPE
Project Report

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE

Computational Workloads for Commonly Used Signal Processing Kernels

5a. CONTRACT NUMBER
FA8721-05-C-0002

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

M. Arakawa

5d. PROJECT NUMBER
2222

5e. TASK NUMBER
06111

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108

8. PERFORMING ORGANIZATION REPORT NUMBER

Project Report SPR-9

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

ESC/XPK

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
ESC-TR-2006-071

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In the course of designing or evaluating signal processing algorithms, we often must determine the computational workload needed to implement the algorithms on a digital computer. The floating-point operation (flop) counts for real versions of the most common signal processing kernels are well documented. However, the flop counts for kernels operating on complex inputs are not as readily found. This report collects the flop count expressions for both real and complex kernels and also presents brief outlines of the derivations for the flop count expressions.

15. SUBJECT TERMS

computational throughput
digital signal processing
complex linear algebra
complex matrix computations
floating-point operation workload

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	None	86	